Mouse-driven menu interfaces for software tools on a bitmap unix system

K.G. Pammett

INRIA

CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(3) 954 90 20

Rapports de Recherche

N° 310

MOUSE-DRIVEN MENU INTERFACES FOR SOFTWARE TOOLS ON A BITMAP UNIX SYSTEM

Kevin G. PAMMETT

Mai 1984

Mouse-Driven Menu Interfaces for Software Tools On A Bitmap UNIX System

Kevin G. Pammett

I.N.R.I.A.
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay Cedex
France

Research Activities

INRIA 1983-84

Abstract: This document reports on research done at INRIA between January 1983 and March 1984 in the area of human interfaces for software tools oriented toward "bitmap" applications. The tools themselves are described, with special emphasis on the parts of human interface that are specific to having menus and a pointing device, the **mouse**. Tools like a font editor, a bitmap interface to the UNIX shell, a generalized display utility for text and pictures, a callable one-line (string) editor, and a UNIX command-language iteration primitive, are described in detail.

This document is a compendium of the reports that were written during a sabattical year by a *Professor Invité* at INRIA, *L'Institut National De Recherche en Informatique et en Automatique*, a computer science research institute located near Versailles.

Resumé: Ce document décrit la recherche effectuée à l'INRIA entre Janvier 1983 et Mars 1984 dans le domaine de l'interface homme-machine pour des logiciels orientés vers des applications sur un écran "bitmap". On décrit les outils eux-mêmes, en insistant sur tous les aspects d'interface qui concernent spécialement les menus et les dispositifs de désignation sur un écran; c'est à dire: la **souris**. Des outils comme un éditeur de polices, une interface "bitmap" pour "le shell" UNIX, un afficheur généralisé également pour le texte et des images, un éditeur spécialisé pour les chaînes des caractères, et une primitive d'itération dans le langage de commandes UNIX, sont décrits en détails.

Ce document est une collection de rapports qui ont été écrits pendant un stage de *Professor Invité* a l'INRIA, *L'Institut National De Recherche en Informatique et en Automatique*, près de Versailles, France.

Keywords: Bitmap, mouse, pointing devices, font editor, Raster-OPs, Human Interfaces, Software Tools, Menus, window systems, UNIX.

Mots-Clés: Bitmap, interfaces homme-machines, souris, éditeur de polices, Raster-OPs, Menus, systèmes de fenêtrages, UNIX, logiciels.

Kevin G. Pammett, INRIA, May 24, 1984.

Final Report On Research Activities

or

"Fifteen Months At INRIA"

January 1983 - April 1984

Kevin Pammett

I.N.R.I.A.
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay Cedex
France

Abstract

This document is a compendium of the reports that I have written over the past 15 months during my "stage" as a *Professor Invité* at INRIA, *L'Institut National De Recherche en Informatique et en Automatique*, a computer science research institute located near Versailles, France.

The first chapter gives some of my overall impressions and tries to bring together some of the various things I have tackled. The remaining chapters are composed of the major documents that I wrote this year, mainly to document the software tools that I worked on. They are included here practically verbatim simply to have everything together in one package.

Table Of Contents

CHAPTER 1

A Compendium Of Papers

Rocquencourt, France.
March 28, 1984.

To: Jean-François ABRAMATIC
Michel GIEN
Maurice ROBIN
Georges NISSEN

1.1 Why This Document

During my time at INRIA I have had the good fortune to be able to **create**. To create ideas, software, and documents. I have also been extremely fortunate in that I have been surrounded by hard-working, intelligent, and very likable people.

This document is addressed to the above mentioned people who made my stay at INRIA possible. Thankyou. I also intend on circulating "my book" to many of my colleagues, and numerous other people, who helped me directly or indirectly this year, in France, England, Germany, Switzerland, the United States, and Canada. This is the primary reason that most of what I have written is in English rather than in French.

Rather than a hand-full of not-necessarily-related papers, I have decided to incorporate the essential papers into this single document. Et le voilà. In my usual style, the papers are fairly long and detailed. As in the past, I call these "working design documents" because the text contains numerous questions intended to further the development of the concepts and software via user feedback. In many cases I also supply a few answers or alternatives to the questions I pose. Afterall, if we had uniquely one answer for each question, or, worse, if we didn't have any questions, then we wouldn't be doing research.

1.2 A Little History

I was invited to do a one-year stage at INRIA in Novemeber 1982, and started here on January 1st, 1983. In October of 83 my contract was extended thru the end of 1984, although even then I intended on taking only 2-3 months of that time to finish up my work at INRIA so that I could start into working on Ada (with Alsys) as soon as possible.

The next few sections are somewhat of an overview of the two projects I was involved with, and to what extent I was involved.

1.2.1 The Sol Project

The INRIA pilot project called **Sol** is a reimplementation of UNIX in PASCAL, for various micros including the HB Level 6, CNET's SM90 (68000), and others.

In the SOL project I was concerned mainly with consulting and helping out the development team in various ways. During 1983 the remaining parts of the operating system were debugged, and I worked with those implementing the higher level tools, like LEX, MAKE, and the SHELL. With some assistence and insistence from me, the SOL **shell** design is upwards compatible with the Bourne SHELL, but incorporates many "C shell" (Bill Joy's CSH) enhancements.

Having been away from UNIX for several years, I took this opportunity to **re**learn C and UNIX V7. I also got to learn Berkeley's Bsd 4.1c and 4.2 version of UNIX which run on VAXes. The documents herein were done on the VAX using UNIX, for example.

1.2.2 ITERation

A "carry over" from my own research work at DEC, the first thing I did at INRIA was to design and implement a UNIX utility which lets users perform command iteration on a standard UNIX system. Having refined and **re**refined ITER chez DEC for 3 years, it was indeed a joy to "start from scratch" and "do it right". I wrote ITER in **C** on a Berkely 4.1c system and ported it to V7 Sm90 UNIX shortly afterwards. Since then I have used ITER daily in my work and continue to find it indispensible. See the chapter on ITER for more details.

This version of ITER has been distributed to Canada, the US, and Europe, over the UNIX uucp network, and numerous people sent me comments on the functionality and user interface. More than ever, I continue to believe that someday **iteration** will receive just treatment within the command languages of current operating systems. Just now, unfortunately, this aspect of the functionality offered to users is sadly lacking; hence yet another version of the command ITERator.

From the beginning I wanted to produce a **bitmap** version of ITER because there one really has the possibility of producing a highly-polished, highly-human-engineered user interface. I also felt that ITER should be made to run in the window system, so as to not have to get into piping input to each subprocess and receiving output (again via pipes) from each command that gets iterated. In BSH (with our latest version of the kernel) I finally found a way around this (scrolling regions) even though I didn't have to time to retrofit ITER.

As an experiment, ITER can converse with users in one of several natural languages, and I worked out a convenient methodology for doing so. The English and French variants of ITER are now in fairly wide use.

1.2.3 Bitmap Tools

Since a new bitmap terminal was being built (by Numelec and CNET) for the SM90, I took on the job of trying to get some software for it. Consequently, I got the sources for MIT's NUnix Personal Computer, and began porting parts of it to the SM90 in June of 1983. I started with the basic primitives, bootstrapped some of the more demanding applications programs (like FE, the font editor), and ended up trying to port their UNIX kernel, too, since it offered the

possibility of having a real windowing system to work from. (But see the chapter called WSYS for further details).

1.2.4 General Consulting

The best part of being an invited professor at INRIA is that I have been fairly free to get involved in numerous projects. Most of this involvement has been document, design, and code review, which has enabled me to have many interesting discussions with the engineers working on projects like: CONCERTO - a software engineering workstation project, IMAGE - the Image processing and Robotics group, KAYAK - some advanced "bureautique", and the NADIR project which is networking, satelites, and communication. I have worked most extensively with the SM90 project which has been gearing up for ongoing UNIX support for "Smx" (version 7 UNIX) on the SM90 (a 68000-based machine).

1.3 VAX/VMS versus UNIX

Now that I have gone back and forth between UNIX and VAX/VMS a few times, one question that is often asked of me boils down to "**which is best**, VMS or UNIX?". I find it an interesting question, and have enjoyed very much the answers that I get when I ask the question back again to those who ask. Invariably, I find that the asker has a strong opinion on the subject.

In my opinion, there is no definitive answer to this question; it depends **a lot** on what you want to do with your VAX. After several years of using a VAX/782 with VMS, I first got access to VAX-UNIX at CNAM, a university in Paris, just after I joined INRIA. There, I was astonished to see how **fast** my C compiles got executed. Really!. There was no point in executing compiles in the background since they'd finish before you had time to do anything else. With 10-15 users signed on, I thought this was a sign of pretty good performance for UNIX on a VAX 11/780... Then, a few weeks later, I discovered that the UNIX engine I was using was **not** a 780 - it was a VAX-11/750, a machine that is supposed to be only 60% of the power of its older brother. And a 782 is 1.8 x a 780!

My real answer to the question, though, is the following: If you want to do the kind of things that UNIX lets you do, like program development, etc, then UNIX is far easier and, in short, **a joy to use**. However, as soon as you want to do something that UNIX was not meant for - like an application where you need to have contiguous files, or programs where you want to use shared memory, then **forget it** for UNIX. As soon as you run into a road block, there's no way out.

On the other hand, almost **anything** is possible with VAX/VMS. You may have to give 15 parameters on your I/O calls and another 10 linker options, etc etc etc; but with VAX/VMS you get there. And once you work out how to do whatever you wanted, VMS will execute it quickly. And the file system is reliable.

I wouldn't want either UNIX of VMS to go away. Perhaps the two can come together some day, but with the diversion that we're already seeing in the UNIX world, it's hard to imagine even the UNIXes coming together, let alone VMS.

1.4 Tools and Human Interfaces

I came to INRIA to work on Software Tools and Human Interfaces. Although I had more like program development tools in mind, in retrospect, the bitmap hardware provided an even richer medium to explore new areas of human engineering and interfaces. Thus, it is the emphasis on the human interface that is the constant thread that runs thru my work at INRIA. Several new areas have been prototyped and described, like an on-line **tutorial** for each tool and a highly-polished human interface (**P**) for finding things in this on-line text.

In the future I intend on building upon this work to make the most of what I have learned, especially in the area of user-tailorable human interfaces, no matter what tool is behind the scenes doing the real work.

1.5 Acknowledgements

This book is dedicated to my wife, **Lynn Pammett**, whose loving support helped enormously to get thru all the trials and tribulations... Let me also thank the following people who contributed significantly to my effectiveness at INRIA.

Jean-François ABRAMATIC
Jean-Michel GUILLOT
Jerome CHAILLOUX
Jean-Marie HULLOT
Michel TOURNOIS
Jean-Claude SOGNO
Pieter VAN DER LINDEN
Marie-Christine PICHOT
Martine CARANNANTE
Jean-Louis BOUCHENEZ
Jean-Lou CLEMENT
Marie-Colette MONET-ERARD
Françoise GUERRE
Fernando FERNANDEZ
Yves DEVILLERS
Christian BAUMHAUER
Jean-Christophe HANUT
Pierre SIMONDON
Michel GIEN
Pierre LEONARD
Mark LOMBARD
Françoise NEVEUX
Warren JACOBS
Michel LOYER
Pierre JANCENE
Carlos KILMANN
Francis LEDRU
Gerard AUROY
Frank BONTEMPS
Gerard OLLIVIER
Christian FIEGEL
Najah NAFFAH
Bernard MAZOYER
Don FRANK
Humberto LUCAS
Gerrard CANY
Maurice ROBIN
Georges NISSEN
Gustave BERRY
Christian MASSON
Marcel POULET
Bruno VERLYCK
Philippe SERMAGE
Brigitte TATARIAN
Louis AUDOIRE
Philippe BAZARD
Ian CAMBELL
Francis MARTIN
Jean-Jacques LEVY
Alain WEGMANN
Gilles KAHN
Bernard MARTIN
Robert JEANSOULIN
Uli FINGER

1.6 Looking Forward

What I really wanted to do for 1984 was to have finished the window system for the SM90, and to work on a tool set and APSE for Ada to live therein... Seems like it's a little too early for all that, but I still think that:

UNIX + a window/bitmap environment + **Ada**

is very definitely a winning formulae for the mid-80s... And I strongly suspect that I'm not alone in that feeling.

CHAPTER 2

Software Tools for A Bitmap UNIX

In Retrospect: From the beginning, I intended to write this document to bring together all of the bitmap tools that I distributed to numerous sites throughout France. I called it "Release Notes", but it ended up being somewhat more than that.

As it turns out, I believe that this is the most important document I produced - at least from a practical day-to-day prospective. The document is (too) large, but it would be **much** worse to have a random collection of little documents floating around to describe things like the font compiler, etc; such things always get lost in the shuffle and you can never find them when you want them.

The rest of this chapter contains the BITMAP document verbatim.

SM90 Software for the Numelec Bitmap Hardware

(Release Notes)

Kevin Pammett

I.N.R.I.A.
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay Cedex
France

2.1 Overall Description

This document describes the collection of software tools that are available from INRIA to work with UNIX on the SM90 as of 27-feb-84, version 3.2 of Smx. Some of these tools have been written from scratch, others have been ported from the MIT system called **NUnix**. The Smx "kernel" support for the bitmap is also described herein, albeit somewhat briefly.

These notes constitute the "release notes" for the tools described herein. See specifically the section "Installing These Tools On Your System" at the end of this document for details relating to directory structure, shell variables, how to obtain documentation, and etc. Since we are somewhat new to the bitmap scenario, there are lots of questions scattered throughout this document. We

hope to get feedback on these issues so that we can further evolve our offering.

2.1.1 A Bitmap Tool Set – A Prospectus

As of this release of Smx, we finally have a coherent combination of bitmap tools, their associated libraries and procedures, and an Smx kernel that matches. We have also begun to "work together" in that "bitmap tool developers" from numerous sites in France have now met and discussed each others concerns. The result of this meeting is a concensus about working together; we have a common purpose and plan to work towards these goals building on top of what one another has done.

This is not to say that all the problems have been solved. Even in this release there are "loose ends" and we didn't have time to put in all the features that everyone wanted. For example, we are distributing the same Raster-Op library that has been available in the past, and we didn't get the window system to work yet. There was a desire to have a "wish list" added to the FE document so that it will someday meet "everyone's" needs; I didn't even get time to write the list let alone make any of the changes. The FE document was printed two full weeks before this one.

We did manage to incorporate a coherent notion of "font file search rules" into **all** of the tools we distribute, though. And there are **hundreds of font files** in the current release instead of the 20 or so that we had before. (We have a program that converts Berkeley format fonts to our ".ft" format). My sincere thanks to Pieter Van Der Linden for both of these long-needed improvements to our offering. (We didn't distribute the font convertor; is anyone out there interested in having it? Ditto for the numerous "font dump" programs we have now accumulated...)

As before, we have "small tools" and "big tools" in the distribution. For the major tools, a separate document describes them in more detail. For the simpler ones, herein is the only written description that we provide. For more details, consult the source listings or contact the author.

The major new bitmap tool this time around is the so-called "bitmap SHELL interface", BSH, for which we give an overview in this document and the details in an ajoining one. BSH features another somewhat novel tool, IED, the pop-up string editor that we document herein (overview) and also in an ajoining tutorial.

Also new is the notion of providing an "**Smx Bitmap Library**"; the repository of all those goodies out there that aren't programs but that users (and us) want to provide to others to build upon. Why is this imporant? Suppose a great "menu package" gets written by someone. (We have one in the works that is also overviewed in this document). Since this is not a program, it is somewhat more difficult to document and distribute, and inherently much more difficult for others to use. We plan to get around this via the **Smx Bitmap Library**. Distributing, documenting, and supporting this library, we hope, will encourage a productive form of working together. If you have developed or plan to develop candidates for this library, please let us know. At this release you will find therein our first version of "font file search rules", "font file loading", and the infamous **sysphs** call, which we have documented (herein) for the first time.

See the section "**The Bitmap Library**", below, for the details of SYSPHS.

A final warning: this release of "bitmap tools" is certain to change. We intend on supporting what we are disributing, but we do not intend on being locked in by it. As we, with help from you, learn more about this problem space, we will evolve our offering in the smoothest way we can engineer. This does **not** mean that we promise upward compatibility from here on in. If you are planning to rely on any of the details of our interfaces or data structures, please confirm this with us first. This warning includes the format of font files which we are seriously considering for an overhaul. Ditto for the Raster-OPs which a lot of people have ideas on evolving in several directions.

2.1.2 How Does The Bitmap Fit In

As of release 3.2 of Smx, the Numelec and its software is much more integrated into the system than was previously the case. Just now things are still somewhat in a state of flux, though, so users should be somewhat aware of the following four very different scenarios in which you can use your bitmap:

(1) **display** - the bitmap screen is used in "standalone" mode by tools like FE, AFT, and P. Here "all of" the Numelec memory is seized by the tool and managed without (almost) any help on the part of the kernel. Since "window prompting functions" rely on some notion of keyboard in this mode, we put your TTY into "cbreak" mode, and hand-do a simulation of the TTY driver to let you enter keyboard data. The backspace character is assumed for deleting the character before the cursor, @ or ^U can be used to delete the whole line and start again, and, in general, entering only <cr> in this mode will cancel the prompt.

(2) **terminal** - if you use the distributed SF command, you can load a font (into kernel memory) and use the Numelec "just like a terminal". This implies that there is some kernel support for treating the Numelec screen and keyboard as "shadow devices" for your console. What this means in detail is described in the section "Kernel Support For The Bitmap", below, but note that you must have the **new version** of Smx for this. Watch out if you have the old version of Smx (as of 21-dec-83) if you try to load a font that is larger than 7000, or if you load a badly-formatted file, you will cause the kernel to write all over its in-core data and there's no telling what will happen after that. We have largely fixed this in the new version but it is still possible to crash the system by loading certain fonts. (I suspect that this has to do with the undocumented handling of the cursor in this mode; the kernel uses the underscore character from the font as the cursor).

(3) **window system** - eventually we hope that users will forget about the "old" modes and simply start using the window system (see the WSYS paper) as a framework for all of their work. Before this is the case, however, the window system itself remains to be debugged. Following that, all bitmap tools will have to be "trained" to live within the conventions of the window system, whatever system we eventually adopt.

(4) **the bitmap shell** - lets you use a specially modified version of the Bourne SHELL which knows about the bitmap, the mouse, and menus, etc. This mode is

half way between modes (2) and (3), above. You have absolutely no multi-window capability whatsoever, but you do have the ability to run arbitrary UNIX processes in a scrolling region of the screen. These processes can do ordinary UNIX I/O to the terminal and the SHELL in conjunction with the Smx kernel provide the notion of a one-window window system.

To get into mode (2), use the SF command, described below. Ditto for the OFF command, which returns you to the mode referred to as (1) above. Mode (3) is not yet available but the WSYS paper is. The bitmap shell (4) is available in preliminary form. It is vaguely suitable for a login shell, but you can only actually do this only if you don't have a .profile in the directory you log in to. Therefore, we suggest you invoke it in the same way as any other "ordinary" command whenever you want to. This mode is probably better to get started with the new interface in any case, because it permits you to go in and out of the bitmap shell without loosing your surrounding environment.

2.1.3 The Raster-OP Library

One of the major goodies that we got from MIT is what we refer to as the "raster-OP library". On the distribution disk, you will find this in:

$BITMAP/numelec/lib/raster/raster.a

with the corresponding sources in the sub-directory, `src'. Almost everything in our bitmap world is linked with respect to this library.

These routines are coded in very efficient 68000 assembly language, and they provide all of the traditional "bit blp" and other raster operations. Due to differences in hardware, these routines are somewhat different from what we got from MIT. Thanks to Jean-François Abramatic for most of the debugging of these routines on the SM90.

Since we have not changed the functionality of these routines, the comments in the source code about how to call the various routines are still valid. In the distribution directory $BITMAP/numelec/lib/raster/src you will find the "*.s" files which were used to create "../raster.a", the actual library. Almost everything in this library is used in one way or another in the various bitmap tools described herein.

In the same source directory you will find ngraphl.s and ngraphl.o, the "graphics routines". For reasons unbeknownst to me this module is not included in "raster.a". Nevertheless, almost all of the tools described in this document are linked with respect to this ngraphl.o module. (In the window system there is a **w**graphla.o module which is analogous to this but performs window clipping, etc, automatically. There is also an **rsda.s** module which is yet another version of "the raster OPs" but which is [supposedly] more suited for being in the UNIX kernel).

SYSPHS was in this library as of our previous release; we will probably leave it there for a while to let people make the transition to the new library as smoothly as possible.

2.1.4 The Bitmap Library – BITLIB

This library in $BITMAP/numelec/lib/bitlib/bitlib.a contains all of the linkable modules that we distribute **except** for the raster-OPs, the idea being that raster-OPs are fundamentally different from things like the "menu package", the "callable editor", and whatever else we find down the road of this genre. Again, the sub-directory `src' contains the sources. One of the source files therein that many users will need is **numelecTIOs.h**, an **include** which contains all of the IOCTL "constant" definitions that you need to do any of the Smx kernel functions described below.

For now, there are just a few things in this library: the **sysphs** memory mapper described below, the "search rules" modules described in the section following that, and some supporting routines like the one that tells you, given a pointer from a font file, whether the corresponding character is actually defined or not. The routines for actually displaying icons or characters from fonts are in the Raster-OP library described above.

We intend on providing MENU and IED here as soon as possible. They are described in the section called "Soon To Come" at the end of this document. In the meanwhile, you can get a feeling for the kind of menu package we are building on by looking at the sources for BSH or P. Even without looking at sources, BSH is an interesting demo of the kinds of things you can do with dynamic and static menus.

2.1.4.1 The Memory Mapper - SYSPHS

In the above-described library you will find the crucial and widely-known **sysphs**, the "UNIX system call" which permits ordinary user-mode programs to map a part of their virtual address space onto the physical memory associated with the Numelec screen. (Before this release sysphs was in the Raster-OP library because that was the only library we had envisioned).

The virtual address that you should use to address screen memory is returned by the sysphs call. This is not the same as the "bitmap = 790000" message that you should see when you boot the new version of UNIX. It is the kernel which finds the Numelec memory (during "autoconfiguration") and then (during sysphs) decides where to map it in virtually.

The details of sysphs are somewhat subject to change and still somewhat of a mystery to us. If you make the "standard" call, as in:

```
screen = sysphs( 0x300, 256, 2)
```

you will map in all of the numelec memory, and its virtual address in your program will be returned in SCREEN. The **2** code is now what I use in most distributed tools. It means "map read/write and make sure the memory is actually there". If you map with any number whose lowest bit is 1, you are mapping for read references only. If the "2" bit is set, you are asking the kernel to make sure the memory exists before it maps it into your address

space. (The wisedom of all this escapes me; I'm simply documenting what I have managed to discover). The "300" is some measure of what segment you want to map, and the 256 is how much of the memory you want to map. If you only want to map the visible part of the memory, 128, here will do.

2.1.5 Font File Search Rules

Now that others have begun making bitmap tools for Smx and the numelec, we have decided to agree upon a set of "search rules" that will be used by "all tools" needing to associate user references to fonts and the actual files in which these fonts are stored. To encourage users to apply the same rules, we have written the needed routines (in C) and now distribute and support them as part of the "Smx Bitmap Library".

Here is the scenario: When any of the well-behaved bitmap tools need to locate a font file, be it to load the font or for **any** reason, these "search rules" are used to find the actual file, given the reference. For example, when you invoke the font editor, FE, as in:

```
$ fe my_font
```

the font you think you are referring to is what you "always" refer to via the string 'my_font'. The applications tools arrange to present you a consistent view of this, without forcing you to know what directories fonts are in, etc. The rationale for this is simply that it's convenient for most fonts to be stored "in the system". Even when you are working with a particular set of fonts, they can still be "in the system" as far as you are concerned. Experience has shown that one rarely wants to keep general-use fonts in the directories where you go about your everyday work.

None of this means that users who want to have their own fonts must put them "in the system", however. The first priority of any bitmap tool should be to take the file reference to be exactly what the user gave, with perhaps the exception of the file type ".ft", which is always defaulted.

The new search rules mechanism is based upon having a sequence of directories to look in. In default, there are two such directories, the current directory, and a fixed one - the font library - which is `/usr/lib/NUFT'. Whenever you make a font file reference in this case, the file is sought in each of these directories in order. Looking for a reference, say `xyz', includes the defaulting of an `.ft' extension if the user didn't already give that in his reference.

To make this mechanism more user tailorable, the SHELL symbol NUFT can be defined, much like PATH, to contain a sequence of directory names separated by colons. When you invoke the .profile at the top of the file tree on the numelec distribution disk, you will see that the NUFT you get is defaulted to:

```
:$BITMAP/numelec/lib/NUFT/:/usr/lib/NUFT/
```

where the $BITMAP has already been expanded in this case. These rules cause the search rule mechanism to look in 3 directories, in order: the current directory, (because of the leading colon), the distribution disk's "font library",

and the assumed existing font library on your system.

Be forewarned that if you define a NUFT in your own environment, just as is the case for PATH, you are completely on your own with respect to defining it properly. The default rules described above apply only when you do **not** have a NUFT. If you do not specifically designate the current directory in your NUFT, font file references will never be taken as you typed them, since this is specifically the meaning of "don't prefix anything".

2.1.6 The Font Library – NUFT

One of the things that most people will notice in the new release is the hundreds of fonts that you will find in:

$BITMAP/numelec/lib/NUFT

The name 'NUFT' comes from the variable that you can define as part of your font file search rules, described above. It stands for **NU**melec **FT** fonts.

All of the files in this directory that have an '.ft' extension are fonts that we have verified. They should work when loaded into the Smx kernel, or when handed to any of our programs like FE, AFT, P, etc. Some of these fonts came from MIT, a lot of them came from Berkeley, and some of them were created by various people "by hand". The levels-of-gray fonts are used in the "image files" demo that you will find described in the section herein on P.

For a simple way to get a look at the fonts supplied on the distribution disk, use the following sequence, which assumes that you have invoked the definitions file described at the end of this document.

```
$ . /mnt/.profile
$ FT=$BITMAP/numelec/lib/NUFT
$ AFT_Sample='La chaine que je veux chaque fois'
$ export AFT_Sample
$ iter -qvs $FT/fontset aft -v $FT/%e
```

In the above, the iterator will come up and propose to affiche the first font, displaying its name in the command to do so. You can answer 'y' to see it, in which case ITER will automatically go on to the next font afterwards. You can also answer 'n' to the "do you want to" question to simply go on to the next font. If you type '*' you will see the Name-SET of fonts that exist. A companion document describes ITER, also known as IT (in which case all user dialogue is in French), in more detail.

Another way to get some feel for the numerous fonts is to use the ZOOM list defined in the .profile in the catalogue DOCs. See the description of this iteration in the section at the end of this document.

2.2 Using The Bitmap As A Terminal – SF, SR, and OFF

With previous versions of Smx as distributed, you could not use the Numelec as an ordinary terminal **or** as the console. With version 3.2 this is somewhat better, but we're still not there. There are at least two outstanding problems: First, when you connect up the Numelec and actually boot the system, the firmware that reads "x;" can actually read this from the Numelec keyboard (although it can't echo the characters), but after the then-produced ">" prompt, the firmware cannot read the typed file name. I do not know why this is or what can be done about it. The second problem is that until full UNIX is running and an SF command done, the software has no way of displaying what you type. You can type blind, though...

As a stop-gap measure, while waiting for the window system, we will provide an SF command which loads a font into the UNIX kernel and turns your Numelec into something like a terminal. If you do:

$ sf name

the font file found using the "search rules" described above will be loaded. If you were not previously using a font, the screen is cleared and filled with a black background as a side-effect. In any case, henceforth the Smx kernel echoes every character typed on the console on the Numelec screen. (If you prefer a the white background, see the SR command described below). For a second and subsequent SFs, the screen is left intack.

Once you have done an SF, if you have another console on your system, the effect is that the numelec and its keyboard become "shadow devices" for your console. That is, typing on either keyboard is indistinguishable, and output always goes to both screens. In this mode, in default, the whole screen is used. That is to say, when it's time to scroll, the kernel assumes that there are 780 raster lines visible on the screen.

However, if you then invoke something like (the ordinary) EMIN, you will see that this "SF mode" is quite inadequate. (We intend on installing a "real" line discipline for handling this in the kernel, somewhat like MIT's "H19 line mode", but the details of this remain to be settled. For now, the hacked terminal driver knows a little about having a font around and displaying characters, but it has no knowledge of ESCs, etc, whatsoever. Berry (at Sophia) has produced a "bitmap version" of EMIN. but it's not clear (to me) that this is anything other than a stop-gap solution. Even if you have this version of EMIN, you will still see the above problem when you invoke anyting else that does not co-operate with the kernel's notion of doing I/O to the numelec. My suspicion is that "real terminal handling" for EMIN as well as for simple tools like LS should be transparent and implemented by the kernel; probably within the framework of the window system. This is an open issue for the present, though.

Most likely, users will want to stop or "turn off" the effect of the SF command, since there are several tools now that manage the Numelec screen by themselves and know nothing about SF and the hacked KL driver.

To "undo" the effect of an SF, then, simply issue the:

$ off

command. This form has no parameters or options and simply resets a kernel variable that tells the console driver to NOT treat the Numelec as a terminal. (This reset is done via an IOCTL invented for this purpose and is why this command doesn't work in the 21-dec-83 version of Smx).

To actually unload fonts from the Smx kernel, you must use the `**all**' form:

$ off -a

which literally unloads all of the fonts that are currently in the kernel. After this form, **no** form of "$ sf nnn" will be accepted. That is to say, your state after an "$ off -a" is exactly the same as it is when you boot the machine. Just now, (unfortunatley), this is the **only** way to actually unload any fonts from the kernel. We plan to improve this situation.

A special form of SF exists in which you type:

$ sf ?

and a list (via LS) of the ".ft" files is presented. Indeed, if you are set up to use search rules based on the shell variable "NUFT", this form of SF is quite handy to verify what you think you've set up because **all** of the directories that your rules cover are reported on. Even if you don't have a NUFT, there is a default and in all cases the directory where the fonts are stored is used, and only ".ft" font files are displayed. Since the shell may obscond the '?' character, it is best to quote it. A "real" SF command might implement this by "-v" or something. Comments?

Note that the "name" form of SF uses the standard rules described above to load fonts. Thus, if you take the default rules, or if you define your NUFT to have an empty field (start with colon, for example), then you will see that:

$ sf mydir/myfont

will always try first for the indicated font in the `mydir' catalogue which is immediately subbordinate to the current directory. This is distincly different from automatically sticking a `/usr/lib/fonts/' in front of the user-given reference. If you are having a problem with your search rules, try the question-mark form shown above since it will explicitly spell out and show you the defaults currently in effect.

It is unwise to try and use font file names that do not end in ".ft"; several of the tools have hard-wired this in, both functionally and literally. We have begun to undo this, but it will be a while before tools like FE let go of their desire to hide the ".ft" but not insist that it be attached to the file name in any case.

No matter what form of argument defaulting you use, if the kernel cannot load the font file, a message is produced and the SF command is ignored. That is, if you have a font established before, you still have it; if you didn't, you still don't.

After you have loaded one or more font files, the kernel keeps detailed information on them in an internal "font table". If you use the (probably temporary) form of SF which is simply "$ sf -", you can see a display of this table. See the IOCTL description for this call for more details.

One thing that is evident from the dump of the font table, though, is that there is an index, **i**, which gets associated with a font when it is loaded. This index becomes the only thing you need to completely specify a font inside the kernel once some fonts have been loaded. To provide a way for users to revert to using a previously-loaded font, the form:

$ sf nnn

is provided. The **nnn** operand must be decimal numeric, and it must refer to a currently-loaded font. If this is the case, the above command is exactly equivalent to a SF <pathname> where the pathname corresponds to the original one given when the **nnn** was assigned.

2.2.1 Scrolling Regions – SR

Much like the stop-gap SF command, there is another "goodie" called SR that lets you effectively issue some of the IOCTLs that have to do with the numelec. The most important of these IOCTLs has to do with "scrolling regions". Normally the Smx kernel treats the full 780 raster lines of the screen as the only scrolling region. However, an IOCTL now lets you change the kernel's notion of this.

The form:

$ sr nnn

exists to let you set the kernel's "screen size" variable (LINESIZE) to the decimal value, nnn. A check is made to make sure that this is not less than 10 or greater than the equivalent of two full screen fulls.

As a related "goodie", if you do:

$ sr

you will be told the current setting of the kernel variable that says how many raster lines there are on the screen. (The number is returned by the IOCTL as its value). Normally this is 780, but mixed-mode tools like BSH set this to less so that it can maintain a menu on the screen even while "standard" commands like LS do transparent I/O to the screen.

Finally, you can issue:

$ sr -b

or

$ sr -w

to set the background default color to BLACK or WHITE, respectively. Note that if you use the -b or -w form of SR, you cannot also give the nnn operand. We will either fix this, or replace SR completely.

These forms are all very experimental; they are simply a "poor-man's" way to

simulate what BSH does.

2.3 Kernel Support For The Bitmap

This is the first release of Smx when we have documented support in the kernel for the bitmap. For any of what we say in this section to work, you must boot the version of unix that is in the numelec/bin directory on the **numelec** distribution disk. (We did not force this version on users that didn't ask for the numelec stuff; in future releases we intend on distribution only one version of the kernel).

All of the support is implemented in terms of IOCTL calls, which are described generally in IOCTL(2) of the UNIX manual. For the actual IOCTLs that we have for the numelec, see the section called "**Smx Bitmap Library**", above, where we point you to the header file that we distribute with all the needed constants, etc.

In general, the support for the numelec is described in two sections. First, for the "raster device" itself there are the following IOCTLs:

BTIO_FONT -This one is "load font". You must give the entire file name to be loaded, and the kernel literally reads this entire file into numelec memory. The loading starts from the top of the memory on the numelec card, and procedes downward. If you give a file name that you have previously loaded into the kernel, the load will **not** be done; the kernel is "smart enough" to realize that this has already been loaded. Note that this check is based on the I-number and device number and **not** on the last-modified time of the file. Thus, if you actually change a font and want to reload it, you must specifically unload the old one first. See $ off -a

When you load a font into the Smx kernel, the underscore (ASCII 0134) is used for the cursor. Just now there is no way to change this. If you find that the kernel is simply not happy with a certain font, look at this character and the space character. Ditto for the DEL character (0177); the printf in the kernel uses this for delays on the console. They both have **very** special meanings. See the FE document for more details.

BTIOLINSIZ -This call lets you set the scrolling region size. See the SR (set region) command for a users interface to this and for more details on the parameters, etc.

BTIO_F_OFF -This call lets you ask the kernel to STOP using any of the in-core fonts for echoing on the numelec screen. No fonts are unloaded, the `off' call simply causes the echoing to stop. You can continue using the font you were using with the SF command.

BTIO_F_ON -Lets you resume after an OFF. This is what the SF command uses for the **nnn** form.

BTIO_F_WHITE -Lets you default the background to white.

BTIO_F_BLACK -Is the default and specifies a black background with white letters.

BTIO_F_CFONT -Causes a printout (literally) from the kernel which displays the font table managed by the kernel. This is **not** a feature of Smx that users should rely on; for now it is there only for our purposes. This will probably become a call that passes this data back to a calling user processes.

BTIO_F_SAVE -Is reserved for future use. It will have to do with providing a way for non-co-operating tools to provide the minimum co-operation that we will need to "co-exist".

BTIO_F_RESTORE -Is the reverse fo _SAVE.

For dealing with the "mouse device" the Smx kernel keeps an in-core data structure which is used to maintain a notion of where the mouse is and what buttons have been pressed. Most of the following IOCTLs let you work with this data structure.

BTIOMINIT -Is the standard way to initialize the mouse. That is to say, after such an init all delta-x and delta-y values returned by the other mouse calls will be relative to the (x,y) values you give in this init call.

BTIOMZERO -Is exactly like the above only you do not give an (x,y) value. What this means in detail is simply that all of the elements of the mouse data structure in the kernel are set to 0 by this call.

BTIOMWAIT -Is the usualy way to ask the kernel to pass back the mouse data. The call does not return until something has happened to the mouse since the previous BTIOMWAIT call. Either a button press or a movememt, however slight, of the mouse will cause this call to return.

BTIOMNWAIT -Is like the above only there is no wait. The call returns immediately and the caller is left to take it from there.

Currently, the standard form of these calls is:

result = ioctl(2, BTIOxxxx [, arg]);

The value returned is normally -1 if there was an error, or 0, otherwise. When the IOCTL has a documented return value, then `result' will contain either -1 (in error) or the returned value.

The **2** given above is indicative of the fact that the numelec IOCTLs don't work unless you are using the console. Internally things are rigged to make output going to the console also go to the numelec screen; if you are not on the

console, then this kludge would not work at all. Therefore the IOCTL mechanism refuses to do anything unless you are really using the console. Obviously we have to fix this, and will.

2.4 The Font Editor -- FE

A companion document describes what I have called **FE**, the font or **icon** editor. This utility lets you edit, create, or merge character sets (called **fonts** or <<police>> en français). You can actually change the form (raster) of characters as well as their "parameters" - the variables that tell the routines that deal with fonts how to paint characters, what offsets to use, and etc.

The font editor is what was called FEDCON or FTEDIT at MIT. Aside from porting it intack to the SM90, the user interface has been changed significantly and several functions (like HELP.xxx, PICTURE, LOAD PICTURE, TUTORIAL, "poubelle", and etc) have been added.

If you are trying to use FE and can't get the "help files", it is probably because they weren't installed in the "right" directory on your system. In a manner similar to how bitmap tools find font files, you can also define a shell variable to "point" to the "Numelec Doc" directory. See the section on "Installing These Tools On Your System" at the end of this document for more details.

Once you have a font, routines from the Raster-OP library are used to paint characters on the screen. See **VidChr**, etc. This is the same way that icons are manipulated: you create them with the font editor, extract them with FE's function **Char Def**, compile the resultant file, and then use **VidChr** with the variable defined, just as if you were painting a real ASCII character. When you use an icon directly in this manner, the icon is referred to by its name; all association with ASCII codes or with the font that it came from is abandoned.

2.5 The Bitmap SHELL -- BSH

A companion paper describes what I have termed the "bitmap SHELL", or BSH. In actuality, this is a bitmap interface to the standard Bourne shell, SH. (On UNIX, the SHELL is the command interpreter or user interface to the operating system). The BSH interface contains a menu package, a mouse interface, three specialized fonts, and more than a dozen icons, all of which are literally linked in with the current version of the Smx Bourne shell, SH.

The interface essentially traps all lines (commands) going to the shell to let you:

(1) **edit** any typed-in command before it is executed.

(2) specify that commands come from a user-tailored **menu**.

(3) **recall old commands** via a HISTORY mechanism.

and much more.

One of the most novel things about BSH is its use of the callable string editor, IED, described in overview below. Also note the very different uses for the menu package: on-screen menus, "static" pop-up menus, half-dynamic pop-up menus (in IED), and completely dynamic pop-up menus, in the HISTORY mechanism.

Due to liscencing agreements with BTL, we cannot distribute all the source for BSH. If you ask for a "source distribution of numelec software", you will get the bitmap, mouse, and interface part, but not the part that actually comes from Bell Labs.

Since there are numerous experimental things in the human interface offered by BSH, we would very much like feedback from users. Is the interface useful? What makes using it difficult? or easy? Do you **really** use it? For what kind of work do you find it **not** useful?

2.6 The Callable String Editor – IED

A companion paper describes what I have termed the "Callable String Editor", IED. This is not a program but rather a set of routines that you can link in with any bitmap application that wants to offer users the possibility of editing a character string "on the fly".

It is IED that lets you edit the last command you typed to BSH, etc. It uses a special, very large font so that you can easily see the string as you edit it. It also makes somewhat novel use of icons and pop-up menus. See the MODIFY mode for a way to edit a string without using the keyboard.

2.7 Display A Font – AFT

The AFT utility exists for the limited purpose of displaying a given font on the Numelec screen. (The name comes from 'Affiche_FT'). Normally, one invokes AFT as, for example:

$ aft -v major_font [other_font_1 other_font_2 ...]

the font file names are treated exactly as they are in the font editor, FE, regarding search rules, merging, and etc.

The -v is the usual way to ask for "the whole story", and the sequence of font files mean the same as they do for the font editor, FE. That is, the resultant font is the one that is obtained by starting with 'major_font' and merging into vacant font slots the characters found in the other fonts given.

The following command-line options apply:

-V -For 'verbose'. This option causes the screen full to contain much more than just the font. For example, the rectangular framework in which the font characters are displayed is drawn with lines dividing each box for each character, much like a checker board. Also, the bottom part of the screen is filled in with various "statistics" about the font, including a "sample string" if you give one.

-T -For 'terse'. This is the default, and implies the opposite behavior of verbose. The screen is filled with only the merged font to be displayed, plus the pathname of the file where the font came from. In terse mode nothing else whatsoever is output.

-C -For 'clear'. Normally, a clear-screen is done when you invoke AFT, and white writing on a black background is the default. The -C option is, for now, just another way to specify the default.

-W -For 'white'. In this case, AFT clears the screen initially by filling it with -1. Henceforth all displaying of the font characters, etc, is done in "reverse video" mode.

-L -For 'Load/merge messages'. Normally, no messages are produced on the console when you invoke AFT. If you give the -L option, a one-line message will be produced that demonstrates the time required to load or merge each font. The first half of the message is produced when the font is about to be loaded; the completion of the one-line message happens after the font is loaded.

In verbose mode there is one more "option" available: the ability to have a sample string displayed along with the font. If you define a shell variable as in:

$ AFT_Sample='this IS a Sample string'

and EXPORT this variable, then the string defined by the variable will also be output using the merged font to be displayed. This is modeled after "sample" in the font editor, FE. If you don't select verbose mode, the AFT_Sample variable is ignored.

AFT displays the font in a manner quite similar to FE, so in theory one could use the font editor to display fonts and we would not need AFT. However, FE is quite slow to invoke, and is much more powerful than is desirable when what you want is to simply display a font. The real reason, though, is that FE insists on being interactive so you would have to wait for the editor to come up and literally mouse your way out each time.

For a "canned" way to look at the fonts supplied on the distribution disk, see the section called **The Font Library**, above.

2.8 The Font Compiler – FtToC

Since a font (file) is a binary (structured) file, you can't normally do anything with a font except to use it with a utility that is expecting this type of file - like FE or SF. The "font compiler", FtToC, helps bridge this gap. You "compile" a font file by invoking FtToC, and it produces a C source file which you can then include in your program in the usual way. By including a declaration for the variable defined in this compiled font, you can then use the variable instead of a font name and henceforth the font is "hard wired" into your program.

The syntax of the "font compiler" is fixed and no defaulting is applied. You must give exactly three parameters, and could invoke it as, for example:

```
$ FtToC file.ft outfile.c VFont
```

The input-font file is 'file.ft', and it must be designated exactly. (i.e. no search rules or ".ft" defaulting apply). The output file, 'outfile.c', is created and can be compiled directly. The result of compiling this file will be to define and initialize the C variable 'VFont'. In your C program you should declare the "compiled" variable as in:

```
extern struct CharDef VFont[];
```

and you then use VFont directly in routine calls where a "pointer to a (loaded) font" is required. See the distributed header file `ft.h' for the definition of CharDef. In essence, it doesn't really matter if the font is literally loaded or not; it only matters that it is there and that you have a pointer to it.

The future of this utility is to be decided. One has to wonder why you need to hard-wire in fonts when this is supposed to be an integrated bitmap system. For now, it's **not** - hence the kludgery. You can load a font into numelec memory, but you can't use this font to put up a string at an arbitrary screen location. The menu package, for example, is completely dependent upon having a font around to write menu labels, etc. Ditto for any "bitmap" tool which has to produce error messages, etc. BSH has three fonts wired-in in this way.

This whole technique is very similar to FE's icon or "Char Def" notion. It's a way of compiling a font instead of compiling a single icon. Either way you end up with an initialized variable that you can use with the Raster-OPs that expect these data structures. See the document that describes FE for more details.

2.9 A Bitmap-Screen Display Utility – P

P is a utility specifically written at INRIA for the Numelec bitmap hardware. Like the utility called MORE in Berkely UNIX, P is specifically intended to let you conveniently peruse (look at) a file, a list of files, or text that comes from standard input. Since user-defined fonts can be specified, you can even use P to display image files (satelite pictures, etc), or, in fact, anything whatsoever. Each byte in the file simply tells P which font character to use when displaying the file.

Unlike MORE, however, P works on the bitmap screen and has been designed to take full advantage of it. For example, you can specify the character font you want to use, or even use a sequence of predefined font names to provide a ZOOM facility. Also you can scroll horizontally or vertically and by lines of text ("jump scroll") or by raster lines ("smooth scroll"). More standard functions let you SEARCH, PAGE, or SCROLL backwards or forwards, and you and randomly move around in the file via "labels" and "GOTOs", etc.

On the numelec distribution disk you will find a sub-directory subordinate to P called `image'. Therein is a .profile which sets things up for you and tells you how to invoke P to see some **satelite pictures** that we display with three levels-of-gray fonts generated using FE. Try the ZOOM command and the NEXT FILE command to look at the two binary images also stored in that directory. You can look at these files with any font, but with anything other than the `gris*' fonts it is likely to be quite non-sensical. Many thanks to Robert Jeansoulin for producing the fonts and for providing the binary images.

A companion document describes **P** more fully. The sources for P are also available, since P was specifically written to exemplify the use of Raster-OPs, using menus and the mouse, etc.

2.10 The Terminal Emulator – TE

We got another goodie from MIT, a program they called TESM or something like that. We have included this in the current distribution package, but therein you will find only the executable form, not the sources.

It was hard to decide whether to distribute TE or not. The pros are that it's very useful; it is used in FE to make the help screens (See the DEMO section, below.) and could be used by others for a similar purpose. It's also useful to get a feel for Raster-OPs and font changing, etc. You can use it interactively or you can prepare files of input for it and simply reap the results. In either case, **no** programming is involved, yet you get to try out graphics, fonts, and Raster-OPs.

The factors against distributing TE are simply that I havn't really worked with it very much and don't really understand the vast majority of the function you can do with TE. I have made a start at formatting the MIT document that "tells it all", but havn't finished this yet. This document is in the Numelec_Doc directory with all the others even tho it's not polished yet.

In the meanwhile, the Sophia group have improved TE quite a lot. The new version can even work with arabic fonts where the default is to write characters on a line starting from the right and going left, or chineese, where you write from top to bottom. Undoubtedly their version of TE is better than the one I use for FE; this is why I didn't distribute my sources for it. Are there others out there who are interested in seeing some form of TE in the INRIA distribution package?

In reality, TE provides us with a framework for breadboarding (prototyping) what we may eventually put into UNIX's bitmap terminal-handling driver. Indeed, some people have suggested that we put a "semi-graphics **line**

discipline driver" into the kernel of Smx, and that what this discipline should implement is very much what TE implements, if not exactly so. The real reason I distributed the TE document was to get more exposure to this idea, so that others could review the details of the escape sequence that are being proposed, etc. If you want to see changes in this area, please let us know.

2.11 A Bitmap Game – The Game Of LIFE

What package would be complete without at least one game? And since Conway's "game of LIFE" is well-known and simple, I used it as my first sample program that displayed something on the bitmap. After I got into interfacing with the numelec mouse, I again used LIFE as my first "application". Finally, it was LIFE who got the first icon-menu, which you get to use to select the "character" that LIFE will use to display each life square that is occupied.

You will find $BITMAP/numelec/life on the distribution disk (if you didn't get an early copy), and therein are the sources, (another starting point if you want to do similar things), the icon font, and several "starting configuration" pictures. To invoke LIFE with one of these configurations, I suggest you use:

$ life -i <chess

where you will find the other names that you can use instead of `chess' in the Name-SET called `configs.set'. The -i brings you up in "interactive" mode, where you will see each successive generation of LIFE (only) each time you hit the LEFT mouse button. The default is for LIFE to simply generate and display the generations (configurations) one after another.

There is a -v command line option, which is the usual VERBOSE (versus TERSE) mode. Furthermore, you can use -B to cause LIFE to start producing a message but continue displaying the generations when a loop in the patterns is detected. Currently the program detects loops up to 17 steps long. (This is a constant in the program; LIFE could be recompiled to detect loops of any length). The asterics command will display the status of loop detection vis a vis the existing generations. The number of generations (17) was chosen so that this display fits easily on a CRT screen. If you give the -W command line option, black characters on a white background will be used instead of the default, white on black. The `R' pop-up command (for reverse) switches you from one of these modes to the other. Finally, the -1 (one) option will cause LIFE to use a solid white square rather than the default (`mec') icon to display the successive generations. (The nomenclature -1 was chosen because LIFE uses the constant -1 to fill the 32-bit words that you see on the screen).

No matter what mode you are in, you can always interrupt LIFE by hitting any mouse button. When you see the "*** interrupt ***" message, you can use the MIDDLE mouse button to bring up what is vaguely like a command pop-up menu. Type `?' there to see the list of available commands. When you leave this mode, use Go or Continue, depending on whether you want to leave in "step-by-step" or "continual LIFE generation" mode.

The other use of mouse buttons in LIFE is to change any LIFE generation, or to specfy the initial configuration. If you hit the RIGHT mouse button, you enter

"hit and destroy" mode where you (a) select a LIFE square by pointing to it. The program will highlight this square. Following that, you (b) press the button pointing to the same square to either create a new LIFE creature (if there wasn't one on that square before) or to destroy an existing one, otherwise. Whenever you destroy LIFE the square is effected with a grey scale to highlight this fact. When you return to normal LIFE mode, the editing box will disappear and the grey scale squares will disappear.

While you are in pop-up command mode, use the F command (for Font) to invoke the icon pop-up menu of Life Font characters. If you select one of these icons it will be used instead of the default to represent a LIFE being in each of the squares where there is one.

A final goodie: you will see that LIFE implements a **direction-sensitive** cursor; it points in the direction you are moving, using one of 8 icons to do so.

Enjoy!

2.12 Curve Drawing and Graphics - LISS

The program called LISS is a curve drawing utility done at MIT for electrical engineers. Also referred to as "lissajous", after the French scientist who invented this notion, LISS is the closest thing that we currently have to "graphics" on the SM90 bitmap. Although I only vaguely know how to use LISS, I intend on putting it into our package of goodies since it serves to demo the Numelec's software emulation of vector drawing. Indeed, even without much of an understanding of the theory behind the kind of curves and equations that LISS uses, I have spent many a happy moment or two amusing myself with the fabulous plots you can conger up.

The following text is largely taken directly from the help text built into LISS. For a reason that I never did figure out, LISS refuses to make a window large enough to actually display the last one third of the text. The root of this problem is that the MIT NUmachine's screen is the same shape as ours but turned sideways 90 degrees.

The scenario is as follows:

Assume you have a number of synchronized function generators, driving the X and Y inputs of an oscilloscope through a number of phase shifters, adders, and multipliers. The patterns generated by these waveforms are Lissajous patterns. Each function generator can produce either a sinusoidal, triangular, or rectangular signal with a certain symmetry (duty cycle), offset (dc component), amplitude, and frequency. LISS is a digital simulation of lissajous patterns.

The screen is divided into four windows:

1 -The top window is where the pattern is drawn. You may initiate drawing the pattern by pressing the left mouse button in this window. The right mouse button will clear this window. The middle button will toggle the 'Sequence' switch described below.

2 -The second window identifies X and Y (the horizontal and vertical displacements) as a function of the waveforms. After selecting the appropriate box in the window, any algebraic expression made of waveforms, constants, and the '+', '-', and '*' operators may be entered. Terms may also be grouped using parenthesis. A waveforms is entered by the letters 'r', 's', or 't' followed by a digit between 0 and 7.

3 -The third window allows for the definition of up to 12 waveforms. First select the box under 'Func' and enter 'r', 's', or 't' followed by a digit. Amplitude is in terms of pixels and can be a fraction (e.g. 1/3), Offset is in terms of pixels, Symmetry and Phase are in the range of [-180, 180] degrees.

4 -The last window is the menu window. Each menu item may be selected, toggled, or modified, as usual. When you select a function, the menu box is reverse videoed so that you know this function is in progress.

On the menu you will see **P**, the number of points in the pattern being displayed. Initially, Lissajous loads the file /tmp/liss/lissajous.sin which contains a table of 15000 points of the sine function. If this table is not found, Lissajous offers to build it (takes about 5 minutes). If you do not have a directory 'liss' in your /tmp, LISS will not be able to create this file at all and it will run **much** slower. When you do have such a file, and when P (the number of points you use -- see the on-screen LISS menu) is set to any factor of the number 60000, Lissajous will not need to create a table for that number of points. The larger this number, the slower is Lissajous!

The following menu-mode functions are available:

Gain -There are two 'gain' functions, **G X** and **G Y**. They let you change the gain for X and Y and are measured in decibels.

P -Stands for 'points' and is mentioned above. If you invoke this menu item you will be invited to input another integral number which will become the number of points to be used in the plot.

Mode -Lets you change the way that points and lines get drawn on the screen. **Mode** may be either Or, Xor, or Points.

Plot -Specifies what functions are plotted. If this menu box contains **(X, Y)** it means that Y vs. X is plotted. Toggling this item will give **Sweep X** and **Sweep Y** which plots either X or Y against a horizontal sweep.

Sequence -The **Sequence** menu box lets you change from a mode where the whole screen is used to one where it is divided logically into four quadrants. If toggled, **sequence** will result in successive patterns to be drawn in half size, at the four corners of the top window. This is so that the patterns may be compared to one another.

Overlay -Provides a way for you to draw one graph over top of another. If toggled, **Overlay** will inhibit clearing of top window.

For those interested in performance, note that the first forms of CLEAR involve approximately half the number of memory references that the reverse video form does. Can you see the difference in perceived speed?

On the other hand, it is easy to see the speed difference between things like CLEAR and almost everything else which uses the Raster-OPs. When you see the screen clear in FE or P, you are seeing a highly optimized version of exactly the same thing that CLEAR does.

2.13.2 Capturing What Is On The Screen – SAVE, FAST, and LOAD

In a manner much like CLEAR, there are three programs which map in the screen and either fill it with data from a file, or save it (i.e. read the screen) into a file. The scenario is as follows:

Once you have prepared a screen full of any type of graphics, text, or whatever, the program SAVE can be used to store away the screen in a UNIX file. The alternate name, FAST, does the same job only much faster. These two programs are file system LiNks to one another.

It is conventional to use the file extension ".raw" for such binary files. These files made by SAVE and FAST are simply binary images of what is in the screen memory. To put back this memory, then, use the program LOAD giving the file name that you want to be literally copied into the screen memory. You can give any file whatsoever here. If you want to see a binary "picture" of an ASCII file or even a program, simply hand these files to LOAD and you'll see quite bizarre (but legitimate) results.

The "help.xxx" functions in the font editor, FE, are examples of what you can do with SAVE and the equivalent of LOAD. I prepare the "help screens" offline (see the DEMO section, below) and FE simply reads the ".raw" files onto the screen to produce the "help". This scenario is (by chance) almost exactly the same as the "pix" function in Lissajous.

2.13.3 LINES – Connect Up (X,Y) Co-ordinates with Straight Lines

The program called LINES lets you simply "connect up" (X,Y) co-ordinate pairs with lines drawn by the VidCnt raster-op routine. If you suddenly want to know where a certain point on the screen is, LINES can be very helpful since is "points out for you" exactly where specific points are.

In default, LINES asks for the co-ordinates of the end points of the LINES from standard input. This lets you use LINES interactively, or you can prepare files with the co-ordinates to reproduce a previous session of LINES.

The command line options described below let you do black-on-white or white-on-black lines. Also, you can draw lines on top of an existing screen or you can have a redraw done when LINES first comes up.

-V -Is the usual VERBOSE notion. LINES will tell you the resulting (absolute) X and Y co-ordinates after you enter each line end-point.

-T -Terse; no messages are produced under normal circumstances. This is useful when you use LINES in a "canned" application, like the ones that build the FE help screens using LINES, TE, and various other programs.

-W -Causes LINES to come up and clear the screen with a white background. Henceforth the lines are drawn with the video mode that makes the lines black.

-C -For Clear. This is the opposite of -W, the default. With -C the screen is cleared and filled with -1 as soon as LINES comes up. The corresponding video mode is also established.

LINES is used in preparing screen fulls of graphics, lines, and text, much like those use in FE's help. See the DEMO section where this is described, below.

2.13.4 More Line Drawing – BOX and RECT

These two programs also demo the line drawing raster-OPs in the library. You call up either BOX or RECT with a parameter, **N**, which becomes the number of pixels left between concentric boxes or rectangles drawn on the screen. If you do not give the **N**, the program will prompt for it from standard input.

A **box** is a square, which starts out taking up the whole square size of the screen. A **rect** is similar except that the entire rectangular part of the screen is taken. Sometimes BOXes or RECTs can be useful as part of making up "canned screens" (see below), or, you can use the programs simply to get a feeling for the speed and effect of using the described Raster-OPs.

These programs have several command line options which let you see the difference between, say, writing in "store" or "xor" or "clear" modes, etc. Just now, the options provided include:

-R -Is for Redraw. If you do not give this option, no clear screen is done before the program begins. The default is to do the redraw with a black background.

-W -Is how you ask for a redraw, and for a white background. This is juse like -R except that the background color changes. This also has an effect on the video mode (VideoM) that we use when calling the Raster-OPs.

-X -Causes the program to affiche the lines with an XOR video mode. If you put up any sequence of boxes or rectangles with this mode, and then repeat this operation, the screen reverts to what it was before the first drawing. Thus is the magic of exclusive-OR.

BOX and RECT are file system links to one another. This is to say, like SAVE and FAST, they are the same program - I simply use the invocation name as another

Save -Lets you leave a session without having to loose all the definitions, etc, that you have made. **Save** stores the state of LISS in a binary file (with extension .lis) so that you can carry on where you left off via the **recall** function.

Recall -Is the reverse of **save** in that **recall** prompts for a file name and lets you reload a state that was saved with **save**. This is used especially in the distribution directory where there are a number of 'demo' files that were created in this manner. To find these files, look in the distributed "$BITMAP/.profile" file for the variable LISS. Therein do an "ls -l *.lis" and you will see the files that you can **recall** once you invoke LISS.

Show -Is analogous to the bitmap tools SAVE and FAST except that they are "built in" to Lissajous. **Show** displays a .pix file created by the Pix function. The entire screen is redrawn.

Pix -Is a function like the bitmap tool called LOAD. **Pix** creates a .pix file which is a binary "copy" of the contents of the screen. The MIT notion of ".pix" corresponds exactly with our ".raw" scenario even though they were invented separately. One difference is that in LISS, this notion is "owned" by the tool and therefore not known outside of it. On the other hand, FAST and SAVE work independently of whatever program actually put stuff on the screen. Ditto for LOAD.

!<cmd> -Lets you enter any UNIX command, which gets executed in a child process. Just now the standard output of the contolling terminal (the TTY) is inherited by the child - NOT the Numelec output stream. When you return to Lissajous, a screen refresh is done if the program that you ran actually changed the bitmap screen.

Refresh -Lets you have the screen redrawn. The function currently being displayed is NOT redrawn; this is really just a way to get rid of garbage on the screen.

Exit -Is the standard way to leave Lissajous and return to UNIX. If you kill LISS and return to the shell via any other means, you will have to "stty echo" and "stty -cbreak" to get your terminal back to what it was.

Quit -Rather than a menu choice, ^**B** is the quit character. It will abort drawing and sine table computation; you do NOT leave Lissajous and can then continue to interact with it.

Char Del -When LISS is prompting you for input, ^**U** will kill this input (from the terminal) and you are invited to reenter the data.

The future of Lissajous within the realm of INRIA is somewhat uncertain. We are not in any way commited to continue to distribute it, and take no responsibility for its use. It does make a wonderful demo, though, so it's likely to stay. If anyone out there wants to develop LISS further, please let us know; we would put improved versions into our offering for the benefit of all.

For a quick glance at using LISS, see the **recall** function described above; the

distributed `demo' files are in the directory for which the corresponding symbol (LISS) can be found in the $BITMAP/.profile definitions file.

2.13 Simple Bitmap Tools – GOODIES

Here we present the very simple tools that were developed for the bitmap more for a demonstration and learning reason than anything. It is also these tools more than the "big" ones for which we want to distribute source. If you want to know about line drawing, for example, a glance at the source for LINES is invaluable. Likewise, for using the MENU package in imaginitive and useful ways, the source for BSH should provide a running head start.

These are mostly the tools whose source you will find in the distribution disk directory $BITMAP/numelec/goodies/src; we distribute this so that others have a place to start at for implementing similar "little tools" for the bitmap.

2.13.1 Fill The Screen – CLEAR

The utility called CLEAR is probably the place to start if you've just got your Numelec and want to see if the hardware has been installed correctly and is working. Otherwise, skip this section; CLEAR is really a tool whose utility is very short lived once your hardware is properly installed and working.

CLEAR is a very simple, non-optimized C program that maps in the memory that corresponds to the Numelec screen, and then goes thru a simple loop to fill each 32-bit word of screen memory with a value. Only the memory that corresponds to the screen is affected. (More than half of the memory on the "Numelec card" is **not** visible on the screen).

If you simply invoke CLEAR as in:

```
$ clear
```

it will clear the screen (black background) because the default **fill value** is 0. Any value whatsoever can be given to CLEAR, though. For example, if you pass on a **hex** value on the command line as in:

```
$ clear 0F0F0FFF
```

this is the value that will be deposited into each of the approximately 25,000 longwords of memory in the visible part of the screen. No matter how you give the number on the command line, it is taken to be hexadecimal. A leading minus sign, as in "clear -1", works just fine, though.

The special form: "clear ~" is somewhat different. Rather than depositing a simple value, this form picks up each word of memory, reverses (compliments) each bit in the word, and puts it back again. The effect is that "clear ~" implements a snapshot form of **reverse video**.

way to "pass on an option" to the program.

2.14 Preparing DEMO Screens

When you use any of the HELP functions in FE, what you see on the screen (the french text) is the result of a simple sequence of procedures that are best exemplified by the following SH script. The parameter passed ($1) is essentially the name of a set of files that all take part in the preparation of the "demo" screen that has that name. (In FE there is a set of these names because there are 6 such HELP screens).

```
$ te <$1.text          --Interpret the text part.
$ lines <$1.lines      --Draw boxes wherever.
$ mouseicon            --Canned programs for special effects.
$ fast $1.raw          --Save the screen in a binary file.
```

The action of the FAST command (described above) is simply to save the contents of the screen (literally!) in a binary file. Thus, you can recoup the same screen, later, via "$ LOAD xxx.raw" where `xxx' is the "name" described above.

The magic of all this is the TE command, used first, above. This utility interprets the indicated file of prepared text mixed with escape sequences. For more, see the description of TE, above.

To help those who might want to prepare such screens of text or graphics, the procedures and text files used for the FE screens are distributed as part of the sources for FE. Look in the sub-directory, `help', of FE's source directory for the goods.

2.15 Soon To Come

FE - needs lots of high-level functions like the ability to grow and shrink characters or whole fonts by a given (x,y) scale factor. I have promised a "wish list" in the FE document but didn't get that done in time for this distribution. During the preparation for the distribution we saw very clearly the need for a DIFF program for font files, and perhaps even an FSCK to help recoup damaged fonts. (Last minute goodie: Peter Van Der Linden has produced a first version of what he calls FCCK - Font Consistency CheckKer - and we are putting this (undocumented) into our ./bin directory. Using it is very much like FSCK).

EMIN - is now vailable in a version that is specifically tailored for the bitmap. For more info, see Berry at Sophia. For the 27-feb-84 distribution of Smx we wanted to distribute this as part of the "bitmap package" but it was impossible. We didn't have the sources of EMIN to integrate the "search rules" library routine described herein for locating font files. Presumably EMIN will be distributed from Sophia with the distributed **bitlib** search-rules modification.

MENU - within the MIT source files we found a fairly good "menu package"; the

one that is now used in all bitmap tools that work with the mouse and menus. Eventually this will be documented and distributed as a separate package; just now you have to "dig it out" and read source listings to find out how to use it. For a glance at the kinds of things you can do with this package, look at P, which shows static and pop-up menus, FE which relies heavily on menu chaining and uses a single pop-up to connect (conceptually) the functions that apply in several modes, and BSH which demonstrates dynamically created menus (HISTORY) which grow and shrink as well as **user** editable menus (STRINGS) and, of course, the standard on-screen function and hidden pop-up menus.

ITER - is described by a companion document and currently is independent of the Numelec hardware. I intend on using ITER as an example of a utility that uses multiple screens and windows, though, so will not describe it further until a "bitmap" version of ITER is available. ITER is a perfect example of a tool that will want to use the one-line callable editor, IED, to provide users with a way to edit the command string that is being iterated. (Currently this is done in a completely non-elegant way by calling up /bin/ed in a subprocess). Even before we have a real window system, ITER could be made to function in exactly the same way that BSH does; since an execution window is already provided by the Smx kernel, ITER has all it needs...

WSYS - is the window manager that I spent about 2 months trying to port from the MIT system. This is already described in a companion document even though the system is not yet available. It seems that there is a bug somewhere in the initialization of the system, so there's really nothing to distribute. Something having to do with initializing the system causes some important kernel mode data structures to get overwritten. Wouldn't it be wonderful to have a real debugger (with SET WATCH_POINT) in the kernel?

TRACK -- Follow The Mouse

I made a start at writing simple filter called TRACK that simply tracks the mouse and passes on (x,y) co-ordinates in a filtering fashion. Thus, you might do something like:

```
$ track | lines
```

to "point out" where you want your lines drawn with the mouse.

There a few issues to resolve with respect to TRACK. First, who should "show" the effect of TRACK on the screen - TRACK itself or "someone else"? Should there be a TRACK option to have it both ways? Second, should TRACK output relative or absolute x and y co-ordinates? LINES, for example, assumes relative co-ordinates. This point of this (TRACK) kind of functional decomposition, however, is to make the producer and the consumer of co-ordinates be independent. However, independence is in jeprody once you start tying together special tools like TRACK and LINES.

Comments?

2.16 Installing These Tools On Your System

This section describes the actual details of installing the various tools that we distribute.

At the very top of the file heirarchy that we distribute, you will find a **.profile** that sets things up for you to test out the distributed system. We suggest the following procedure for you to first test out what we distribute before you actually copy any files at all:

```
$ /etc/mount /dev/pdxxx /anywhere      --make the disk be known.
$ cd /anywhere                         --get yourself there.
$ . .profile                           --define a new environment.
$ cat README                           --to actually do the installation.
```

and now you have a new PATH, a new NUFT, and etc.

No matter where you mount our disk, if you change your current directory to that directory and then invoke our profile, you will have a symbol $BITMAP which specifies the part of our pathnames that is due to where you installed the files. There will also be other symbols that "point" you to the source directories where each of the tools are generated.

To get a quick glimpse of the fonts delivered with the system, use the AFT utility as shown in the example in the section, **The Font Library**, above.

The font editor, FE, uses the shell variable **Numelec_Doc** to locate the HELP files. This variable should **not** point directly to the directory or file that is used for help. Instead, the bitmap tools build in the notion of their own directory structure, and the user can provide, via Numelec_Doc, the "root" of this structure. The details of this are at the end of the FE paper.

2.16.1 Bugs or Limitations

Since these are "release notes", some discussion of "bugs" appears throughout this paper. Here are a few more which are worth noting separately...

(1) There is no way to "multiplex" the mouse and keyboard notions of I/O. For now, you simply cannot have your cake and eat it too. We intend on implementing a new SIGNAL within the Smx kernel so that an applications program can issue a READ to pick up characters typed on the keyboard, but then get interrupted (as is the case for all sys calls in UNIX) when there is mouse activity. This is the traditional approach to this problem taken by MIT in their NUnix and by numerous others.

(2) The **rewrite** function in FE often fails if the font is "too high". Do a "refresh" and the screen should be redrawn correctly every time.

(3) The string write routines in FE are unable to handle fonts (like arabic) which should be drawn from right to left on the screen. Others have pointed out that the FE mouse icon is in the "wrong place" when you have

characters that get drawn from the right.

(4) In the version of Smx that we actually distribute now, it is difficult to **undo** a SET FONT (SF) command. The OFF command is distributed, but OFF simply tells the kernel to no longer use the police. It remains in memory, though, so that you can revert to using it again via "$ sf nnn". We do not have a general memory manager in place in the kernel to actually permit the unloading of a police. This is why the kernel may refuse to load any fonts after it reaches the maximum number (20) or the maximum memory (on the numelec card) allocated to font files. It is also why we added the CLEAR FONT call described in the "Kernel Support" section in this paper.

(5) It is not unheard of to "miss interrupts" from the mouse. Especially, for example, in the P function called AUTO, you will see that the system only "looks for button pushes" between raster OPs. I suspect that this is inherent in the fact that all versions of UNIX except the real window system under development do **not** use real interrupts to look at the mouse device. If the clock dispatcher is "fast enough", you're safe. Otherwise you simply have to push the button again.

(6) Users have noted problems when using any of the bitmap tools from a terminal other than the console. We will look at this in the context of the window system where we will have to decide if "bitmap tools" have a life outside of WSYS. See the comments on this in the section: "Kernel Support For The Bitmap", above.

(7) Previously, it was the case that you could crash Smx by loading certain fonts. Now, we believe this is fixed - if you get the Smx that we distributed after the first of March. If you do have trouble like this, look at the definition of the space character; **space** must defined and its overall size must be appropriate for the **entire** font.

2.17 Further Information

For more information on using the Numelec bitmap hardware with the SM90 system, contact the author of this paper or Michel Loyer at INRIA. In general, INRIA will distribute these tools in binary form to those doing research. The issue of distributing the source for these tools is not completely settled; in general we have and will give sources to those who want to work **with** us. We have not ironed out all the details of this yet with MIT but indications from them are that there will be no problem.

All documents that have to do with the Numelec software are available in printed form or on-line. You can get a printed version of this text by simply copying the ASCII file:

$BITMAP/numelec/Numelec_Doc/bitmap.doc

if it was put there on your system; otherwise ask the person who installed the Numelec software or contact the author.

2.17.1 Documents Available On-Line

Here is the Name-SET of documents that you are invited to look at with P in an iteration as part of the `demo' package for bitmap tools. Once you have invoked the .profile at the top of our file heirarchy, you will have a symbol, `DOCs' which you can use with CHDIR to get you to the directory where the following Name-SET is stored.

```
:
: $DOCs/docset.set -- the set of on-line documents for BITMAP tools:
:
bitmap.doc          --this document; an overview of the rest
fe.doc              --the Font or Icon Editor
p.doc               --a bitmap file afficher like MORE
bsh.doc             --the bitmap SHELL
ied.doc             --callable string editor
wsys.doc            --the (future!) window system
iter.doc            --the SHELL command iterator
te.doc              --an H19 (etc) terminal emmulator
:
: end of docset.set
```

While waiting for an even more-automated "demo package", you can peruse your choice of the bitmap documents via:

```
$ iter -qvs doc.set p
```

which will call up P for any document that you answer 'y' to. To look at the names of the documents in the actual Name-SET we distribute, type the ITER query-mode command '*' (asterics). To leave the iterator, type `q' (for Quit). In you invoke the context (.profile) that goes with DOCs, you will see a specially-prepared ZOOM list for P. Therein we have defined some 15 fonts that are all more or less appropriate for looking at the documents we distribute.

Any suggestions on how to improve this software or this text would be appreciated.

Contact: Kevin Pammett, INRIA, Rocquencourt. (3) 954-9020 poste 3160.

[end of BITMAP tutorial text -- updated as of 21 March, 1984]

This page left blank for double-sided copying

CHAPTER 3

A General-Purpose Bitmap Font Editor

In Retrospect: The font editor from MIT was the first significant thing that I made work on the bitmap, and porting it to UNIX (the SM90) convinced me that one **really can** save an enormous amount of time by porting software as opposed to rewriting it.

My sincere thanks to MIT, because FE provided me with an excellent opportunity to experiment with changing the human interface and trying out "little" but quite significant visual effects. Rather than having to concern myself with the real programming inside of a font editor, I was freed up to do the kind of research I came to INRIA to do.

The rest of this chapter contains the FE document verbatim.

FE - The Numelec Bitmap Font Editor

(Tutorial)

Kevin Pammett

I.N.R.I.A.
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay Cedex
France

3.1 SYNOPSIS

FE [-v] [-t] [-b] [-l nnn file[.pic]] fontfile[.ft] [mergefile[.ft] ...]

Note: in this document I use the nomenclature **FE** *to refer to the font editor. On the UNIX system distributed by INRIA, the actual name that you type is "fe" (in lower case). Another convention is that in this document I use upper case to name functions, like EXIT CHAR MODE, while in FE per se the names in the various menus tend to be mixed case for aesthetic reasons.*

This is a tutorial document for FE since it describes how to use the utility. Moreover, this is a "working design document" for FE because the text contains numerous questions intended to further the development of FE via feedback

from readers. This version reflects FE at "baselevel 3", which coincides with version 3.2 of Smx, and supercedes all previous versions of this document.

3.2 DESCRIPTION

FE is an interactive fonts editor which allows the user to create new font files, and to edit or merge existing ones. The actual form of any character can be changed, and characters can be assigned to arbitrary ASCII codes. The FE "system" comes complete with documentation and several dozen fonts.

FE is a version of MIT's FEDCON (or FTEDIT) editor, adapted for the SM90 UNIX system (called 'Smx') and the Numelec bit-map hardware by Kevin Pammett of INRIA. This document is a rewrite of Bern Niamir's distributed version (April 1982). It is available on-line while using FE via TUTORIAL or the "A propos" functions. These functions let you peruse this text in a flexible way. The former enters via the PAGE function, the latter simulates an initial SEARCH, prompting for the string ("context") you want to read about.

3.2.1 Command Line Description

The simplest way to invoke FE is to give the name of a font file, as in:

```
$ FE myfont
```

this invokes FE in interactive mode, with the font whose name is 'myfont'. For the actual correspondence between font names and their files, see the section "Font File Search Rules", below. If you invoke FE without giving a file name argument, a "Usage" message will be produced and FE exits.

Ordinarily, FE enters interactive mode as soon as it has interpretted the command line, although there are ways to use the font editor in "batch" mode. No matter what the mode, the following command line options are available:

-v -The **-v** (VERBOSE) and **-t** (TERSE) flags control the level of messages, etc, that FE produces as you go about the various functions. For now, at least, the default is VERBOSE.

-t -Terse. If you don't like all the "doing I/O..." (etc) messages, invoke FE with a **-t** option or toggle the same switch via the VERBOSE function available on one of the menus.

-l n f -Load Picture. This option lets you specify an ascii code and a picture (in FE's LOAD PICTURE sense). The operation is exactly as if the user had performed the interactive counterpart except that it's all specified on the command line. The **n** operand is a decimal number, and the **f** operand is the name of the picture file. For more details, see the section "Loading and Merging Fonts", below. Note also the interaction between this and the -b flag.

-b -Batch. After the command line has been interpretted, FE normally enters interactive mode automatically. When you give a **-b** option anywhere on the command line, this option is not taken. Instead, if any changes have been made to the font under edit, an automatic "write" is done, and FE returns to the UNIX command line. This option is primarily intended for use with the -L (load picture) command line option for the automatic creation of fonts where the characters have been generated via pictures and the pictures have been generated by programs.

The option characters can be given in either upper or lower case, but they cannot be grouped together as in `-bv'; instead use, e.g., "$ fe -b -v name ...".

3.2.2 Loading and Merging Fonts

In general, the "editfile" and "mergefile" names given on the command line can be any legal UNIX file names or font file names; the standard "font file search rules" are used to locate these files. The default file name extension (suffix) is ".ft", and the SHELL symbol "NUFT is consulted much like PATH. If a "mergefile" does not exist, it is skipped and no message is issued. On the other hand, if "editfile" does not exist, it is created, again, without a message being issued. This is how you create a new font "from scratch"; simply invoke FE as in:

$ fe newfont

and a check is made to see if `newfont.ft' exists. If such a file exists but the format of it is not verifiable as a font, the file reference is rejected and you cannot edit this "font file". However, having searched all the "NUFT" directories, FE comes back to the original reference and explicitly creates a file with the needed "skeleton structure" whenever such a file does not already exist. In general, it is after to start with an existing font file, copy it, and use that as a starting point in creating new fonts. This helps avoid problems of not having the space character defined right, etc.

It is useful to think of FE interpretting its command line, and acting on the options as they are encountered. You cannot invoke FE without giving at least one font file name. In the case where only a font file name is given, there is no real "interpretting" to do except for the loading of the font so that it can be edited.

If a subsequent font file name is encountered on the command line, it is merged with the initial font. This means that characters are copied from mergefiles only if the corresponding slot in the font under edit is not being used. The effect of this is that FE "picks up" characters from the sequence of fonts that are given on the command line. In all cases, a **single** resultant font is obtained; it is this set of characters that you are presented with when FE first comes up interactively. If indeed the font you edit is different from the first name given, that is, if any merging or LOADing of PICTUREs is done, FE annotates the font file name with "modified"; this is equivalent to the user actuallly having edited the primary font. See QUIT for why this is relevant.

Mixed in with merge file names on the command line, you can give the LOAD

PICTURE equivalent, as in:

```
$ fe fontfile -l 32 myspace -l 123 special_pic
```

which would bring up FE in interactive mode (because no -b was specified) editing `fontfile.ft' in which characters 32 and 123 (decimal) had already been loaded from the indicated picture files. Note that the name you give here is the file name **without** the ".pic" extension. This is typical of FE. Again in this case, the editor marks the file "modified" when it comes up interactively.

For a non-interactive example of using FE, assume that you had a set (file) which contained the nnn (ascii codes) and corresponding file names, as 2-tuples on each line of the set. You could invoke FE repeatedly as in:

```
$ iter -qvs pic_set fe -v my_font -l %e -b
```

which would completely build the font called 'my_font' using the pictures defined in the set. The -b flag causes FE to write out the file and return to the shell immediately after interpretting the command line.

We should provide the "inverse" of this function - the ability to extract all or some of the characters in a font as PICTUREs. Comments?

3.2.3 Font File Search Rules

By design, when any reference is made to a font, it is not usually the case that the actual name of the file that the font is stored in is given. This frees up users from having to know the actual location of font files, the fact that these file names end with ".ft", and etc.

In order to present this issue of "font file search rules" consistently, I have written about it in the document that describes the whole tool set available from INRIA that work with the Numelec bitmap. For FE's usage of fonts, suffice it to say that FE follows the rules described in this other document except that when a font file look up fails, FE will create the font for you when you give this name as the primary edit file name.

3.3 An Overview of Font Editing

FE is menu-based; the user selects the desired function from a box at the bottom of the screen or from a pop-up menu which appears and disappears when appropriate. FE uses the mouse for menu-selection and for changing the form of a character, pixel by pixel, if you wish. The keyboard is used for numeric and character input during a font edit session. Input from the keyboard is noticed only when you are invited to type via a "prompt window". Otherwise, the only teletype activity is DEL which you can freely use to exit FE brutally. The editor does put the terminal into "cbreak" mode when you use it, but this is restored automatically when you exit.

FE allows up to two background fonts (or "master" fonts) to be scaled and superimposed on the font that is to be edited. A background font is a font which already exits and which is specified during the editing session. Whenever you enter character edit mode, the corresponding character from the background font is displayed on the screen, superimposed on the primary font character, while the user edits the primary font character. (The primary font is the font which is being created or edited). The background font may be scaled to the size desired for the font being edited or created. The background font is used as a guide for creating the new font.

FE will also create a 'statistics' file containing useful information about the primary font. Since the directory associated with the "stats" file is always the same as that from which the font was loaded, you will notice that when editing a font not found in the current directory, your "stats" files will also end up in this "foreign" directory. Later on in this document each of the fields produced in the statistics report are described in detail. You can look at such a file while in FE by using P and the "SH Command" facility in FE's pop-up menu.

The mouse is used for entering font data information during editing and for selecting options from the menus. In general, the left button on the mouse can be pressed to select options from any menu. In reality, **any** button will do but you must be pointing to a menu. The other mouse buttons tend to change meaning and are described below.

Whenever it is valid to use the mouse buttons, a bull's eye or arrow cursor will appear. When you see the question-mark cursor, take it to mean that you are in "no man's land" and that if you **do** hit a mouse button, it is likely to be ignored. See the "Help.MENU" function for a more complete description. See also the more detailed description of "bitmap screen presentation conventions" in the document on P.

3.3.1 The Mouse ICON

At all times when using FE, you will see an icon on the screen that is meant to tell you what the mouse buttons mean. Except for the names of the functions, this icon is always the same and is meant to symbolize the Numelec mouse with its three function buttons. NOTE: the meaning of the mouse buttons change frequently as you go from one mode to another, or even during the operation of a specific function which does not switch modes.

This is the same mouse icon that you will see in P and other bitmap tools. Like all other icons, it was produced by FE as a simple character (without the character strings) and demonstrates how large a character you can make. (It is 91 x 120 pixels; the max is 128 x 128 pixels). It is our intent to promote this mouse icon as a standard; it should be on the screen at all times when one is using an interactive bitmap tool.

3.3.2 ON-LINE HELP

Two distinct types of "help" are available while you edit. The simplest one appears as videotext-like screen fulls; press one of the six "Help.XXX" functions to get some clues as to what is relevant at the particular place you are at while editing. For example, the "Help.MAIN" function gives you an overview of the font editor; "Help.PIC" tells you about pictures and "char defs", etc. Every menu produced by FE has such an associated help screen. These are the ONLY documents available that are written in french.

Two other functions are available to let you get at the FE TUTORIAL: i.e. this document. If you invoke the TUTORIAL function, you get into a mode much like P or the Bsd MORE utility, intended to let you peruse the tutorial guide for FE. The function called "A Propos" is also available to help you find things in this document. Doing TUTORIAL and then SEARCH is equivalent to entering the tutorial in the first place via "A Propos". The "tutorial" facility was used for the basis of the distributed "P" command, which elaborates greatly on this facility. See "Further Information" at the end of this document for more info.

You will note that, like P, the left-hand button is available while you are in the tutorial system to cause a pop-up menu to appear. In FE, though, this is the FE pop-up menu, not the one from P. This provides a way to invoke some of the FE functions while you are in the tutorial reading about them.

3.3.3 Interactive Font Editing

Although FE works in a limited way in "batch", it is primarily an interactive editor. If you do **not** give the special command line option (-b), FE enters interactive mode automatically after it has interpretted the command line. You will notice that the initial FE cursor flashes. This is to let you know that the editor is there and ready for you to start moving the mouse. After the first time you make contact with the program, the cursor stops flashing never to do so again.

Once in interactive mode, FE has two modes: **font** edit mode and **character** edit mode. Each mode has its own set of menus, and you can switch from one to another easily. There is also a pop-up menu which you can use from either mode to invoke functions that are less frequently used, or, in some cases, functions which are very frequently used from **either** mode like escape-to-shell.

Initially, FE is in **font** edit mode. In this mode all 128 characters of the font are displayed along with some strings of characters to show them in context. If you select (by the RIGHT mouse button) a displayed character, you will get into character edit mode and may edit that character.

In **character** edit mode, FE displays the character being edited in a large "edit box" or window on the screen. It is this image of the character that you manipulate to change the form of a character. Outside of the edit box, the character is also shown in real size and in double size simultaneously with instant feedback while the user is editing or creating the character. Note,

however, that if you turn off the CONTINUE function, this update is not done. (This is what CONTINUE is for!). The background font(s) character(s) are also displayed (optionally) during character editing sessions. See the SHOW and BOARDER functions for more details.

Before detailing further these two modes, the following section introduces some terminology. The menu functions available in each of these two modes are ennumerated and described about two pages below.

3.4 TERMINOLOGY

The following terms are used in this document and in FE itself.

Font -A "font" or "character set" is a data structure, normally stored in a binary file, that defines the form and characteristics of a set of characters meat to be displayed on a raster device. In technical terms, a font is a set of CharDefs; see the appendix (of my WSYS paper) which describes ".ft" format files in detail.

Verification -When a font is loaded, there is a sequence of checks that are made on the contents of the file to ensure that this realy is a font. Pointers are checked to see that they really do point to "reasonable places", min- and max- size variables are checked to verify that they are within certain bounds, and etc. Verification is done by the Smx kernel when a "load font" is done (via the SF command), and by any of the bitmap applications programs (FE, P, BSH, and etc) when they use the Smx bitmap library "load font" mechanism.

Mini-Raster -The bit raster containg the character shape in a .ft file.

Primary Font -This is the font under edit during the operation of FE. It is the first file name you give on the command line when you invoke FE. You cannot change this name without leaving FE, but see the SAVE FONT command for ways around this.

Character Matrix -That portion of the mini-raster that is bounded by a box delineated by the HSize and VSize parameters of the character in a .ft file.

Bounding Box -An imaginary rectangular box that surrounds the on-bits of the character matrix (Note: by the .ft definition, the bounding box is wholly contained in the character matrix. It is not required that the bounding box be the same size as the character matrix.)

Centering Matrix -This is the rectangular box by which the foreground and the background are scaled and centered w.r.t. each other. The centering martix is either the bounding box or the character matrix of the foreground and background characters depending on the status of the Back Cntr switch

(see below).

Adjustment -Refers to the process of reducing the character matrix to become equal to the bounding box. When you invoke FE, the "adjust" box is normally highlighted. This is because the "adjust" function implies "please do adjusting for me", and this is the default. No information is lost if the HSize, VSize, HAdj, and VAdj are updated accordingly. See the "Re-Adjust" function for how this applies to that function.

Adjusted Font -Is a font where all the characters (except possibly for the space character) has been adjusted. The purpose for adjustment is to save memory, since the area between the character matrix and the bounding box is always 0 and may be eliminated. Sometimes it is desirable to have unadjusted characters in the font. For a start, the space character must be unadjusted otherwise the default line height of the font which is VSize of the space character will be lost. (The default space width of a font is the HInc parameter of the space font.) Also, programs that write a character on the screen by bitblt, may not erase the screen beforehand and hence will require unadjusted fonts in order to erase inter-character spacing.

Base point -A point where the cursor is and the character is to be printed in relation to. The relative distance between the top left of the character matrix and the base point is given by HAdj and VAdj. (Here we don't mean a "real" cursor; rather we mean the (x,y) position that is defined internally by the raster-ops and is used implicitly with many of the "character display" operations.)

Increment Point -The raster point where the cursor is updated to after the character is printed. Its distance to the base point is given by HInc and VInc.

Baseline -A horizontal line that passes through the base point.

Pop-Up Menu -A menu, much like the usual on-screen menus, which let you select functions by pointing to the menu box you want and clicking with the mouse. A pop-up menu is different from an ordinary one in that it must be "called up", either by invoking a function or by hitting a mouse button that is reserved for this purpose. Also, pop-up menus disappear immediately after use; you cannot make them stay around permanently on the screen.

Poubelle -(or gargabe can). A circular ring of 8 icons that appear on the screen in font edit mode. Each time you invoke an FE function that would otherwise destory a character matrix, a copy of the character is stored in the poubelle so that you can later recoup it if you want to. This includes the obvious functions like DEPOSIT and PICK UP, and also the EXIT return from character edit mode and the LOADing of a PICTURE on top of a character.

The poubelle icon signifies exactly where, in the ring of poubelles, the next destroyed character raster will be placed. It is expected that this poubelle area will serve as a raster-operations "accumulator" once FE has functions to perform operations on whole rasters, like concatenate, etc.

Picture -An ordinary ASCII file that corresponds to a character matrix in that each on-bit of the matrix appears as a "1", and each off-bit appears as a space. You can create pictures, edit them with any ordinary editor, and then generate font characters to correspond to the new picture via the LOAD PICTURE function.

Since there is more to a character definition than just the raster, you will see "comment" lines in picture files. These are the lines which start with the colon character. Just now these lines are written out faithfully when you do the PICTURE function, but they are studiously ignored when you do LOAD PICTURE. My main reason for this is that I don't want users to have to update this detailed information; if one forgets and the editor believes these "comments" during LOAD PICTURE, the results could be catastrophic. It is a bit of a shame that one looses all this information, though... Comments?

Message Window -A pop-up window which is used to display an error or informational message to the user, normally in reverse video. These windows are triggered to go away after a certain internally-defined time delay. On an unloaded machine, this time is about 2 seconds.

Prompt Window -Like a message window, a prompt window is used by the editor to give the user a prompt and a place to type from the keyboard whenever some character or numeric data is needed by an FE function. Unlike a message window, this type of window will stay on the screen until the user hits carriage return, having entered his answer to the prompt.

If you enter <cr> in response to a prompt window, the function that you invoked which caused the prompt is always aborted. This is a consistency rule that ought (at least) to hold true for all bitmap applications.

Search Rules -The sequence of rules that are applied to a user reference to a font, to actually arrive at the real pathname of the corresponding file name. See the section: "Font File Search Rules" for more details.

3.5 The FONT EDIT Mode

In font edit mode the entire 128 characters in the font are displayed in real size. A menu, called the font menu, is also displayed on the bottom of the screen. In this mode, it is possible to rearrange the position of each character in the font. A character may be picked up by pointing the cursor and pressing the MIDDLE mouse button. This character will replace the cursor and may be moved around. As a side effect, the font character that you pick up is removed from the font, and a copy of this definition is put into the on-screen "poubelle" (garbage can) for later use.

If you do not want a picked-up character to be deleted from where it was, simply deposit it (see below) immediately after picking it up - this will leave the character in the cursor AND also in the font.

Once you have picked up a character, it may be deposited into any character location by pointing to a new position and pressing the LEFT button. The deposit function always takes the cursor as its source, thus, if you deposit when the cursor is the arrow icon, (which represents the null character in this context), the effect is to delete the indicated character from the font. Similarily, if you pick up the not-defined character, this has the effect of deleting the character that you may have picked up (copied into) the cursor. In font edit mode, a special icon (white square) is used to indicate that there is no character assigned to this slot in the font. See REFRESH for another way to "clear out" your cursor.

Besides displaying all the characters in the font, the font character matrix can be used to select specific characters for editing. Hit the RIGHT button while the cursor is pointing to a character in order to edit that character. Just now, the "edit" function cannot be applied to a poubelle icon. If you want to edit something that has ended up in the poubelle, pick it up and deposit it somewhere in the actual font.

The following entries appear on the font menu:

Write -Writes the font on disk making a permanent copy. You do not get to give a file name; the complete pathname of the primary font (the font under edit) is **always** used implicitly. In VERBOSE mode, a message tells you how many bytes were written. After you do a WRITE, the "(modified)" annotation disappears and you can issue a QUIT and leave the editor without interference from FE. For a more flexible function related to WRITE, see SAVE FONT on the pop-up menu.

Note that if you issue WRITE from the pop-up menu and are in character edit mode, you are NOT saving the font with the character that you are in the middle of editing. The same semantics apply to SAVE FONT; see below.

Refresh -Cleans up the screen and redraws it. This function has the side-effect of destroying any character that you may have had in the cursor. Use the "poubelle" to regain any character lost in this way.

Help.MAIN -This function puts FE into a mode where the screen is filled with french text that describes the overall characteristics of the font editor. There is no scrolling, etc; once you hit ANY mouse button you return to where you were.

There are several "Help.XXX" screens available while using FE, in fact, every menu has one. The model here is that each menu has an associated explanation, and the user invokes this function because he wants to know what is relevant given that he is using FE in the mode that is associated with this menu.

For a very different kind of help, see the function "tutorial" or "a propos", both of which appear on the pop-up menu no matter what mode you are in.

Stats -Writes font statistics into a file whose name is the same as the font under edit, but whose extension (file name suffix) is ".stats". The directory associated with the "stats" file is always the same as that from which the font was loaded. Thus, when you are editing a font not found in the current directory, note that any "stats" files that you create will also end up in this "foreign" directory.

The following parameters of each character are reported on:

Code: the character hexadecimal (ASCII) code
HSiz, VSiz, HAdj, VAdj, HInc, VInc: the corresponding six parameters of the character as defined by the .ft format
HRas, VRas: the width and height of the bounding box
HBas: the horizontal distance between the base point and the left of the bounding box
HNBs: the horizontal distance between the right of the bounding box and the increment point
VBsB: the vertical distance between the base line and the bottom of the bounding box
VBsT: the vertical distance between the baseline and the top of the bounding box
Ratio: the ratio of VRas/HRas
Fill: the ratio of on bits in the bounding box to the area of the bounding box.

These parameters are helpful in designing fonts so that they conform to similar parameters of an already existing font. They also help out to verify that you have created exactly what you think you have.

In VERBOSE mode, a message is produced to tell you that the stats file was created ok and what its pathname is. You can look at this file on your terminal with CAT, or on your Numelec screen with P. Likewise, since it is an "ordinary" ASCII file, you can spool it to your line printer, etc.

Adjust -If this switch is toggled on, then each time you enter character edit mode FE will eliminate extra blank vertical and horizontal raster lines in the font in order to reduce the size of the font. Note that this adjustment happens only on characters that have been selected for editing, and that it only happens when the "adjust" box on any menu is highlighted. Note that for some characters in conjunction with some programs, blank raster lines are required in the font because of their erasing action. See also the "Re-Adjust" function for its connection with this.

Bgrnd 1 -The first background file name is provided here. You cannot use any background functions in character edit mode until you provide the corresponding background font name via this function. If you turn off this function (by selecting it with the mouse when it is already reverse-videoed), the indicated font is unLOADED and the space it was using is released.

Bgrnd 2 -The second background file name is provided here. See above.

Quit -Tries to exit FE without writing the font under edit. Once modified, the annotation "(modified)" is added to the top line in font edit mode. When this is the case, the first time you select QUIT a message is issued and you are not allowed to leave FE. To really do an exit at this point, simply select QUIT again immediately. If you select any intervening function, you must again do 2 QUITs to get out. Selecting pop-up does not count as another function here; if you have to do pop-up just to get QUIT, you can still leave the editor via the "2 QUITs" rule.

Menu 2 -Will display another menu in font edit mode, covering the previous menu. These menus are completely independent of those in character edit mode.

Menu 1 -Will return to the first menu of font edit mode.

Menu 3 -Will make the last menu of font edit mode appear.

Char Def -After invoking this entry, any character in the font map may be selected subsequent to which FE will create a C compilable '.c' file of the character in terms of a Char Def (see the .ft format) structure. This is used for including special characters such as icons in source files. All of the icons in FE are generated this way, including the very large "mouse help" icon that appears on every screen.

Wndw Cntr -Causes centering of foreground in the edit window of character edit mode to happen according to the character matrix rather than the bounding box of the character.

Back Cntr -Causes centering of the foreground to the backgrounds to happen according to the character matrix rather than the bounding box. I.e., causes the centering matrix to switch from the bounding box to the character matrix, and vice versa.

Tutorial -This function provides the user with an on-line way to peruse the document that describes FE, without having to know where the document is stored. If this function does not work on your system, see the "Further Information" section at the end of this document. There are certain files that have to be in certain places for all of FE's "help" functions to work.

The first time you do TUTORIAL during an FE session, you are positioned to the top of the document, and you can simply read your way thru it with PAGE or SCROLL, etc. To find information on a particular topic, use the SERCH command. Finally, you can exit this "mode" with the "Quit HELP" function which is assigned to the right-hand mouse button. The next time you reenter via TUTORIAL, you are returned to the screen-full that you had when you left.

For another way to use TUTORIAL, see "A Propos" in the section where we present the character edit mode functions. Both TUTORIAL and "A Propos" appear on the pop-up menu, too, as they are not necessarily associated with either of the FE editing modes.

VERBOSE -Is a toggle function to switch you back and forth between being in "verbose mode" and "terse mode". In VERBOSE, which is the default, lots of pop-up window messages are produced whenever I/O activity or other such critical things are going on, so that the user is assured of the effect of what he has initiated. The TERSE mode is meant for those who know what is going on and don't want to be bothered with "lots of extraneous messages".

Help.MENU -Is a screen full, in french, which describes using the mouse in conjunction with menus and other types of "selecting". Also described are the various icons that make up the FE human interface.

PICTURE -Is a function which lets you indicate a character from the font map, much like Char Def. You also give a **name**, like Char Def, the difference being that an ASCII "picture" of the character is written into the file whose name is `**name**.pic'. The model is that you then edit this file at your leisure to delete lines or columns, to double the size of lines, or whatever. Once you have manipulated the picture with your favorite editor, you use LOAD PICTURE to recoup the character form.

PICTURE files can also be generated by programs - or, in fact, by any means. This is how we created the levels-of-gray fonts that are used by P to display satelite pictures or images.

LOAD PICTURE -Lets you bring in from the outside world the "form" of a character to be added to the font map in the location designated (by the mouse). A 'picture' file is one that was created by PICTURE - **or** you can generate them by any means whatsoever.

There are comment lines in picture files that are there for the

future enhancement of FE. Just now, they are completely ignored by the LOAD PICTURE function. FE leaves the base point as (0,0) and sets the increment point to (0,rhs) where rhs is the right-most bit on the loaded character matrix. Thus, if you produce a character font where you want the characters output side by side with no horizontal or vertical space anywhere, simply doing LOAD PICTURE suffices. If you want something more hand-tailored, you have to enter character edit mode and reset the base and increment points "by hand" after each LOAD PICTURE. You can either delete or simply leave the lines in a picture file that start with a colon (':') as they are just comments (for now).

See the FE command-line option **-L** for a "batch mode" way to interact with LOAD PICTURE. You don't always have to "do it by hand".

Help.PIC -Is a screen full, in french, which describes the notion of PICTURE and picture files. Also described is the icon notion provided by the CHAR DEF function.

! <command> -Lets you enter any UNIX command, which gets executed in a child process. Just now the standard output of the contolling terminal (the TTY) is inherited by the child - NOT the Numelec output stream. The cbreak terminal characterics are restored for the child process, and put back for FE afterwards. Also, if the child process changes the bitmap screen, an automatic redraw will be done by FE when you return. Thus you can "nest" bitmap tools on top of one another without interference.

The "escape to shell" command also appears in various other menus so that you can do this escape without changing modes. For the same reason, it also appears on the pop-up menu described below.

3.5.1 The Pop-Up Menu

For ergnomic reasons, all functions available in FE are not always displayed in a menu which is always on the screen. Instead, only the commonly-used ones are always visible, and a pop-up menu is provided to make the others available. This is done using the SM90 menu package, so that a common interface is presented to users. The pop-up menu function is a direct copy of the same function in P.

In FE the situation is a little more complicated than it in is P, though, because the font editor presents a totally different interface to users depending on what mode you are in. In FE, the pop-up menu is not specifically associated with either **character** edit mode or **font** edit mode; it works in both modes. Most of the functions presented in the menu work in either mode, and do exactly the same thing in either case. Other functions, like REFRESH, do the same thing logically, but are completely different depending on what mode they are invoked from. A few functions are presented no matter what mode you are in, even though you are prohibited from invoking functions like EXIT CHAR MODE

when you are not already **in** that mode. Should the pop-up menu itself change depending on modes? If so, how can the menu package be designed in such a way as to handle this, but **avoid** having to duplicate the part of a menu that is common to several modes?

Another difference with respect to these fundamental modes of FE is that in **font** edit mode the pop-up function is invoked from the on-screen menu (because I didn't want to take away one of the buttons in that mode), while in **character** edit mode there is a button (the left-hand button) reserved for it. In this mode it's more like P. The button that I "took away" in **character** edit mode is the one that used to be assigned to the "Continue" function. This function still exists, it's just that now you have to invoke it directly from the character mode function menu.

No matter how you invoke pop-up, the menu is presented, the mouse icon changes to tell you that all buttons now mean SELECT, and the editor waits for you to make your choice, following which the menu is made to disappear and the screen is restored to what it was before. Then and only then is the function you selected invoked, and following that the mouse icon is put back to its original state unless the function you chose causes a mode switch or terminates the editor, etc.

The pop-up menu is produced in the vicinity of where you were pointing in character edit mode, while in font edit mode it appears at a fixed place on the screen. After this point, **any** mouse button can be used to select from the pop-up menu, and if you SELECT anywhere outside the menu, this is taken to mean that you no longer want to select a pop-up function.

For consistency sake (with other bitmap tools), the first function below is specifically (and only) associated with the pop-up menu. For reasons of not taking up valuable screen space, the following functions are also **not** on any of the other function menus.

Cancel Pop-Up -If you actually select CANCEL POP-UP, the pop-up menu disappears and you return to FE, just as if you had not invoked the pop-up function. There is no difference between selecting this function and moving the mouse outside of the pop-up menu and then pressing any mouse button.

Scale **X** -Is a real number used for changing the scale of the background font along the x axis. When a new character is being created, it's horizontal size will be the horizontal size of the first background character times this scale factor. Used when a new font is being created which is a desired ratio of another (background 1) font. In VERBOSE mode an automatic "SHOW SCALE" is done if you change the value of Scale_X.

Scale **Y** -Is a real number used for changing the scale of the background font along the y axis. When a new character is being created, it's vertical size will be the vertical size of the first background character times this scale factor. In VERBOSE mode an automatic "SHOW SCALE" is done if you change the value of Scale_Y.

Note that the X,Y scale factors are **only** used when the

character that you are editing - the one from the primary font - is not already defined. Any time you edit a character that is defined, the scaling needed to make a background character appear in the same bounding box is always automatically calculated.

Show Scale -Is a simple way to find out what the current setting of the X and Y scale factors is. It is this function which is invoked automatically in VERBOSE mode after you change either of the scale factors.

Save Font -This function writes out the current font, exactly like "WRITE" does, but you are prompted first for the name of the file that FE creates. If you enter <cr>, the function is cancelled. Otherwise the entire font under edit is written out using the name given except that FE suffixes this with ".ft", as usual. In VERBOSE mode, a message tells you how many bytes were written and the actual name of the file that was created.

NOTE that using this function does not change the name of the font under edit; indeed, the SAVE FONT name is forgotten by FE and you have to reenter it if you do a subsequent SAVE FONT. Likewise, the "modified" notion is associated only with the primary font. If you do SAVE FONT, the primary font is still modified and you cannot leave FE without doing QUIT twice.

Another thing to note is that if you do either WRITE or SAVE FONT from character edit mode, it is the font without the modification in progress that is written out. To incorporate the changes you may have made to the actual character under edit, you must return to font edit more before you do the WRITE or SAVE operation.

There are other functions on FE's pop-up menu, but they are described in the sections of this document where they also appear on function menus.

The pop-up menu facility in P and FE has been designed as part of the SM90 bitmap tool set, and will be available as part of the standard "menu package". If you have ideas on what should be provided in this package, please let us know.

3.6 CHARACTER EDIT Mode

The second FE menu, referred to as the character edit menu is used when editing a specific character within the font. Font statistics for the primary and background fonts are displayed at the top of the screen. The top row of the table are parameters of the foreground font, and the next two rows are the parameters of the two background fonts. If you do not currently have any background fonts, the corresponding table rows are simply not displayed.

The following parameters are displayed:

HScal, YScal -For the first row, which describes the primary font, they are the desired size of the bounding box of the foreground. For the next two rows, they are the ratios of the size of the bounding box of the foreground to the background. These parameters may be changed by selecting the appropriate box and entering an integer or a float.

HCent, YCent -For the first row it shows the number of pixels to a grid box in the edit window. For the next two rows, it depicts the horizontal and vertical displacements of the center of the centering matrix of the foreground with respect to the background, again in terms of screen pixels. All of these parameters may be changed by selecting the box.

HSize, VSize -These are the ".ft" parameters of the font, i.e. they are the maximum dimensions of the character matrices for each character of the font when viewed as a whole.

HAdj, VAdj -The location of the basepoint with respect to the top left of the character matrix. These two parameters form the vector used to begin painting a character.

HInc, VInc -The location of the increment point with respect to the basepoint. These two parameters form the vector used to position the cursor after having painted the character on the screen.

HRas, VRas -These two boxes give the dimensions of the bounding box, in pixels. These values change dynamically as you edit a character if you cause the bounding box to grow or shrink.

Also appearing near this region on the screen is the annotation "Char = 0xxx" in which the ASCII code (in octal!) for the character under edit is displayed.

In character edit mode there are two menu displays. Most of the primary functions appear in the first menu; you have to switch to an alternate one to get at the less used one, however. The left-hand mouse button is another way to invoke functions as this calls up the pop-up menu described above. For the meaning of the other mouse buttons in this mode, see "Painting versus Point-By-Point", below.

The following options make up the character edit mode menus:

Exit -Exits to the font menu making changes to the font buffer. It is at this "exit" point that the changes you may have made to a particular character are actually effected with respect to the current font. Just before writing over the original copy of this character in the primary font, the copy is moved to the poubelle just as in all other circumstances where something is about to be destroyed.

It is because of this notion of the changes not taking effect until you EXIT that SAVE FONT and WRITE do not write out these changes-in-progress if you invoke them from character edit mode.

Quit -Returns to font mode without making changes to the buffer.

A Propos -Is simply an alternate way to "enter" the TUTORIAL system. You are invited to type a character string into a prompt window, and the next thing you see is a screen-full of the tutorial, with the line containing the string highlighted with some lines of context on either side. If this is NOT the text you were looking for, simply hit REsearch. Note that each time you invoke "A Propos", FE always starts from the TOP to search for a line containing this string. This is explicitly different from the TUTORIAL function with respect to leaving and re-entering.

The tutorial system is very much like the P command except that it is (much) less sophistocated due to the fact that I don't want to increase the size of FE as much as adding P directly would cause. Probably I should work out a way to make P be literally callable as a subsystem. This would allow an intelligent and consistent form of on-line help for all bitmap tools.

For another way to use "A Propos", see TUTORIAL in the section where we present the font edit mode functions, above. Both TUTORIAL and "A Propos" appear on the pop-up menu, too.

Rewrite -Rewrites the sample string on the screen to reflect the new changes to the character being edited. Note that this has nothing whatsoever to do with writing out or saving the font under edit.

There are currently at least two bugs in this function. (1) if the characters are very high or very low, only every second REWRITE draws them correctly. (2) If your **base** point is to the right of your **increment** point, you will see an overwriting effect rather than the right-to-left writing that you probably wanted.

Refresh -Redisplays the screen.

Sample -Allows the user to enter the sample string which is displayed at the bottom of the screen. This string is always displayed using the font under edit - including the character under edit even though this character is not otherwise considered part of the primary font until you EXIT character mode.

Base -Lets you set the base point, which is the point on the baseline where the character starts. (Technically, the base point is the (x,y) point where the cursor is positioned before any character is printed). After hitting Base, point the cursor near the grid location you want the Base point to become. In VERBOSE mode you will get a guide message about this.

If you change the base point, you will see the resultant change of the (HAdj, VAdj) and (HInc, VInc) points in the table at the top of the screen. This is because the increment point changes its point of reference when you change the base point.

Increment -Lets you set the base point for next character, or, putting this another way, INCREMENT lets you specify where, relative to the base point, the cursor is left after the character is painted. In VERBOSE mode you will get a guide message about establishing a new increment point.

If you change the increment point, you will see the resultant change of the (HInc, VInc) points in the table at the top of the screen. Changing increment never effects any change in the base point; likewise, changing base doesn't change the effect of the increment point, but it may change the actual factors involved in calculating increment.

Continue -This function is a toggle switch between continuous and discreet painting modes. When CONTINUE is not highlighted, the actual-size and double-size versions of the character under edit are **not** updated in real time. You continue to modify the character in the edit box, but must do a Refresh to have the rest of the screen updated.

Show -Show forground character (primary font) in the edit window. If you do not have SHOW highlighted, you cannot do any editing of the foreground character.

Show 1 -Toggle switch between show/no show of background font 1. This character is displayed with a level-of-gray color, guaranteed to be different from that used for the second background font character.

Show 2 -Toggle switch between show/no show of background font 2. This character is displayed with a level-of-gray color which is again different from that used for the first background font character.

Border -Show border (base point increment point, lines around the character which indicate the space taken up by the character) for the foreground font.

Border 1 -Show border for background 1 character.

Border 2 -Show border for background 2 character.

B 1 Char -Sometimes it is desirable to read in another character for the background rather than the same character as the primary character. Selecting Back Char will allow a different character to be entered for the background 2 character.

B 2 Char -Same as B 1 Char for background 2.

Adjust -Same as Adjust in Font edit menu.

ReAdjust -This function provides for a particular form of manipulating the character matrix interactively. First, select any of the four boundaries of the character matrix; the selected boundary will be highlighted. Then select the new position where the

boundary should be extended to. In this way you can add to the raster of a character without being forced to turn a bit on. Note that you cannot "extend" the boundary inside the bounding box; i.e. you cannot take away any raster points this way. After changing one boarder, you are invited to change another. Thus, READJUSTMENT is automatically repeatable; you exit this mode by hitting the right-hand button at any time. In VERBOSE mode you will see message windows that guide you thru this process; in any case the on-screen mouse icon changes to let you know what buttons to use for what.

Menu 2 -Replaces current menu with another for more commands.

Menu 1 -Returns to the first menu.

Multiply -Is part of a basic calculator function provided by FE. This allows the user to multipy integers and reals to deal with scale factors, etc. Should we extend this to provide a whole "calculator" subsystem?

Divide -Divide integers and reals. See Multiply.

Help.CHAR -Is a screen full, in french, which gives an overview of what character edit mode is - including the meaning of the mouse buttons. For more see the "Help.MAIN" description.

Help.FUNC -Is a screen full, in french, which describes the main functions available in character edit mode. In effect, I have abandoned this in favor of TUTORIAL. For more see the "Help.MAIN" description.

3.6.1 **Foreground** versus **Background FONTs**

The purpose of having background fonts is so that you can use known images to guide you in creating new ones. To this end, one or two background character images can be displayed in the edit window WHILE you edit the foreground character. The background characters are displayed with different levels-of-gray so that you can tell one form from another.

The following commands govern the way which the foreground is centered in the edit window:

Window -Recenters the character and its backgrounds to fit squarely in the edit window. I.e., it will cause the center of all three bounding boxes to coincide.

Wndw Cntr -Similar to font edit mode.

In the table at the top of the screen in character edit mode, you will see various values. By selecting row 1, column 3, 4 of this table you can enter the number of pixels that you want to see in each grid box.

The following commands govern the way which the foreground and background are centered and scaled with respect to each other:

Recenter -Will cause the center of the matrices to coincide and all centering offsets to be set to zero.

Reset -Will recenter and window the fonts and initialize all parameters as when character edit mode is entered.

Back Cntr -Similar to font edit mode.

You can also effect this change by selecting any of the boxes in row 1, column 1, 2, and rows 2, 3, column 1 through 4 of the character mode top-of-screen table.

When you edit a character which is not defined in the primary font but for which a character does exist in backgroun font 1, the editor uses the "Scale X" and "Scale Y" values that you can specify "once and for all" (via the pop-up menu) to determine the size of the background character to display. Once editing this null character, all you have to do is "paint" the new character by using the scaled background character as a guide. Note that the grid shown in character edit mode is always the grid which is appropriate for the foreground character. This is why you must decide (based upon your own notion of aesthetics) for each grid box whether "enough" of this is covered by the background image to warrent turning on this bit for the foreground character. Unfortunately, for now, this is the easiest way to make a new font which is a desired ratio of an existing font; we should automate this function based on a best-guess heuristic.

3.6.2 **Painting** versus **Point-By-Point**

In order to edit a character, put the cursor in the edit window. Pressing the RIGHT button down and dragging the cursor will cause bits in the touched grid boxes to be turned on. The middle button causes the bits to be turned off. The right button will toggle the continue box in the menu. While a character is being edited, its parameters in the parameter table will be updated in real time. Also, the real-size and double size character below the edit window will be updated in real time as long as the CONTINUE menu-box is highlighted. See the description of this function for more details.

All foreground parameters displayed in the top row of the table are updated in real time as the character is edited.

In order to create a foreground character that is a KNOWN scale of a background, select the appropriate box in rows 2, 3 and columns 1, 2 of the table and enter the scaling factor. If you are starting from an undefined character and the scaling factor with respect to the first background font is KNOWN, then enter the scaling factors via the pop-up menu functions provided for this purpose. However, if the scaling factor is not known, rather, the ultimate desired size (bounding box) of the character is known, then enter these parameters in row 1, column 1, and 2 of the table. This third mode of operation is the most useful way to create a new character that is a scaled version of the backgrounds. In either case, FE will scale the background

characters and superimpose on them a grid such that once the appropriate bits are turned on, the bounding box of the foreground will have the desired scale.

Integers entered in these boxes will indicate to FE the final raster size of the primary character by which the two backgrounds should be scaled. (HScal and YScal on the first row are the desired horizontal and vertical raster size of the character; i.e., the ultimate width and height of the bit pattern in the centering matrix.)

Additionally, the following boxes may be selected:

HCent and YCent on the top line are the number of pixels for the grid in the edit window. Reducing this will shrink the patterns and the grid in the edit window.

HScal and YScal in the two last lines of the table are the scaling factors of the two background characters with respect to the primary characters. These figures are the ratio of HRas and VRas of the primary to the background. These boxes may also be selected if it is desired to independently scale the backgrounds.

HCent and YCent is the horizontal and vertical offsets of the center of the backgrounds with respect to the center of the foreground. If changed, it will cause the backgrounds to translate to the new location.

3.7 Further Information

For more information on using FE, see the various Help.XXX screen-fulls that are indicated in the function menus of character and font edit mode. These files are stored in ".raw" form, however, so cannot be printed in hard copy. You can get a printed version of this text by copying or printing the ASCII file:

/numelec/fe/help/fe.doc

or by contacting the author. There is a link to this file in the usual "../Numelec_Doc" directory where "all" the bitmap documents are stored. For FE's use, on some systems, the above pathname is not the complete pathname of where the "*.raw" and ".doc" files are stored. If you define and export the shell variable before you invoke FE, as in:

```
$ Numelec_Doc=/mnt
$ export Numelec_Doc
```

then the string "/mnt" is prepended to the "fe.doc" pathname above by FE. This document is the same one that you can peruse while in FE via the TUTORIAL or A PROPOS functions.

Any suggestions on how to improve FE or this text would be appreciated. Contact: Kevin Pammett, INRIA, Rocquencourt. (3) 954-9020 poste 3160 or 3442.

[end of FE tutorial text -- updated as of 21 March, 1984]

CHAPTER 4

A Menu-Driven Bitmap-Screen Display Utility

In Retrospect: I wrote **P** initially just to get my "feet wet" with respect to using fonts to display text and images, and to use the menu package and the mouse, etc. Over the past few years I have forumlated several models of how users want to look at more text on a screen than will fit, and I wanted to see how coherently I could provide for **several** of these models simulaneously. I personally think that MORE has a great deal to learn from P.

The rest of this chapter contains the P document verbatim.

P – A Menu-Driven Bitmap-Screen Display Utility

(Tutorial)

Kevin Pammett

I.N.R.I.A.
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay Cedex
France

4.1 SYNOPSIS

P [-vtez] [-f fontname] [file ...] [-s [string]]

Note: in this document I use the nomenclature **P** *to refer to the bitmap-screen display utility. On the UNIX system distributed by INRIA, the actual name that you type is 'p' (in lower case). Another convention is that in this document I use upper case to name functions, like SEARCH, while in P per se the names in the various menus tend to be mixed case for aesthetic reasons.*

This is a tutorial document for P since it describes how to use the utility. Moreover, this is a "working design document" for P because the text contains numerous questions intended to further the development of P via user feedback. This version reflects P at "baselevel 3", which coincides with version 3.2 of Smx, and supercedes all previous versions of the document.

4.2 DESCRIPTION

P is a prototype tool specifically for the Numelec bitmap hardware. Like the utility called MORE in Berkely UNIX, P is specifically intended to let you conveniently peruse (look at) a file, a list of files, or text that comes from standard input. With tailored fonts, P will also let you display "image files", pictures, or binary files of any sort.

Unlike MORE, P works on the bitmap screen and has been designed to take full advantage of it. For example, you can specify the character font you want to use, you can scroll horizontally or vertically and by lines of text ("jump scroll") or by raster lines ("smooth scroll"), you can SEARCH, PAGE, or SCROLL backwards or forwards, and you can randomly move around in the file via "labels" and "GoTo", etc.

In effect, P is an experiment in using menus and the mouse, and it serves as much to demonstrate the new bitmap hardware and software as anything. The "auto" function seems to demonstrate that we will need to dedicate a 68000 just to handling the screen, a fact that we are not the first to deduce...

4.2.1 Invoking P

The simplest way to use P is, e.g. `$ p fichier.c`
but you can also use it in a pipe as in: `$ (cat text; ls -l; cat end) | p`

In the above examples, P comes up displaying the first page of text read either from the named file, or from the indicated pipeline. A default (built-in) font is used to display the text.

In all cases, the following command-line options apply:

-s [xxx] -Search. This simulates an initial search, exactly like using the menu-search option - except that the screen is NOT drawn until the first match is found. If the first file does not contain the string, the **-s** is ignored and the screen is filled with the first "page" of the file. In either case, an initial search is always taken to mean that the **motion** key default is reSEARCH.

The **-s** option can be given before the **file** command line argument if (and only if) the form "-s xxx" is used. That is to say, if you default the search operand, you must put it last. This means that either you don't use the file argument, or that you put the file name before the **-s** option on the command line. In short, if a **-s** is used, the following argument, if it exists, is always taken to be the "xxx" parameter.

-v -Currently the default, this option causes P to produce numerous messages about what functions you have invoked, what actions this caused to happen, etc.

-t -Terse. This reverses the "verbose flag" and causes P to be much **less** chatty. You can also go back and forth between verbose and terse via the menu function VERBOSE.

-f name -Font. This option is one way to specify a default font to use for what P displays on the screen. You can also change the font in use **during** the session via SET FONT, and see also the ZOOM command for other font change manipulations. For details on the mapping between font file "names" and their actual pathnames, "Font File Search Rules", below. If you use -f and the font can be loaded, a -z on the command line will be ignored.

-z -Zoom. Normally, even if you have set up the needed shell variables for using the ZOOM function, P does not chose one of these fonts as the default. If you invoke P with **-z**, however, an initial ZOOM will be simulated before the first screen-full is displayed.

-e -End-Of-File. If you give this option, P will read the first file entirely into memory before it displays the first screen-full. This is the **default** behavior when P is invoked to read its text from standard input. Note that the -e flag applies only to the first input file. After that you must SEARCH, use a motion function, or use BOTTOM to get to the end of file.

The command line option characters can be given in upper or lower case, they can be specified separately as shown above, or they can be grouped together as in:

$ p -tzs abc file1

which displays 'file1' on the first-found occurrence of the string 'abc', beginning with the first font from the "ZOOM list". The initial mode is terse. If the 'abc' argument had been left out, above, P would come up reading standard input and searching for the string 'file1'.

If you do not give a file name on the command line, P reads from standard input, regardless of whether you actually use it in a pipe. Thus P is like CAT(1). Warning: it's the usual UNIX scenario if you invoke P in the background and then use any P function which needs to read from standard input.

A very useful way to use P is in a pipeline, as in;

$ ls -l | p **&**

where it is LS which produces the text that P lets you look at. In this model, P is seen as a consumer of text; it gobbles up a text stream and then presents you with a menu-driven highly-specialized human interface for looking at this text.

Whenever P is invoked with its input coming from standard input, there is a potential conflict between prompting functions in P, which want to read standard input, and P itself which is continually reading standard input whenever it needs more text. To avoid this situation, P **always reads all of the**

input before prompting when input is coming from standard input.

4.2.2 Multiple Files

It is possible to manipulate several files at once with P, but see the section "Commands For FILE Manipulation" below for a description of the model P uses. Indeed, P lets you give a file list, but the user model is based around treating them sequentially as file units rather than as a simple sequence of lines. Thus, if you invoke P with several file names, as in:

$ p file1 file2 file3 ...

it is only 'file1' which will be read first. After that, you must use the NEXT FILE menu function to continue on to the next named file. Just now there is no way to change the names of the files you gave on the command line, but you can see these names via the SHOW FILEs command. Likewise, you cannot jump from file to file; only NEXT FILE is provided to traverse the text. (Note, though, that you **can** invoke any command from P using a pop-up menu function. Thus you can recurr P with a new list of files).

4.2.3 Windows, Messages, the Mouse, and the Cursor

The first thing you see when you use P is that the screen is taken up with menus, icons, and etc. To take some of the mystery out of it, this section defines the standards and conventions used by P (and other bitmap tools).

When you first invoke P, you will see that the cursor always appears at a fixed place on the screen, and that it starts out flashing. This is specifically to tell you that P is ready for you to use the mouse and invoke commands. As soon as you touch the mouse the cursor stops flashing never to start again. This is to say, once you are in control, P lets you know this by **not** flashing the cursor.

In general, the cursor follows the mouse and its form changes depending on whether you are in the region of a menu or not. To let you know when you can SELECT from a menu, the form of the cursor is a right-pointing arrow. As soon as you move away from a menu, a bulls-eye cursor is displayed.

The main menu takes up most of the bottom part of the screen and is described in more detail in its own section below. You can select a function from either this or the pop-up menu (described near the end of this document) by pointing to the function box and hitting **any** of the three mouse buttons. Within any menu, all three mouse buttons are considered equivalent; outside of a menu they are almost always distinguished.

To the right of the function menu you will see an **icon** that is meant to tell you the meaning the mouse buttons when you are not in a menu. Shown in the icon are three triangles meant to symbolize the three mouse buttons. Follow the dotted lines from these buttons to see which function names are currently associated with each button. This is the same icon that is used in FE and in the bitmap SHELL (BSH), and I intend on promoting it as a standard. During the operation of P the phrases attached to the buttons are likely to change radically.

The right-hand mouse button is generally reserved for leaving P and returning to UNIX. That is, it is the equivalent of the pop-up menu function QUIT. The middle button is the **motion** button that is described in the section called "The motion Functions", below. This button is P's model for how users traverse or "go thru" text.

The left-hand mouse button is normally called SELECT and is the one that we say you should use in a menu to SELECT or invoke a function. If you hit this button outside of the main screen menu, another menu of functions will appear and you can select one of them as is described in the section "Pop-Up Menu Functions" near the end of this paper.

If you make some sort of error while using P, a message is usually produced in a pop-up window. In general, this window is reverse videoed so that you will take notice of it. A time delay is built in to P and short of recompiling, there's no way to change this. The actual time such a window is displayed changes depending on how loaded the machine is. This is something we should probably change. Some people have suggested that we take away the window whenever you move the mouse; this has the nasty side effect of leaving you in the dark (about what happened) if you unintentionally move the mouse even very slightly. Perhaps we should put up a special cursor **inside** a message window and then flush the window either after a fixed real-time delay, or when the user moves the cursor outside of the window, whichever happens first. If we were clever, we could even restore your cursor to where it was before the pop-up window appeared... Comments?

A different user-feedback model is seen when P puts up an information window, as in most of the SHOW functions. Here we have no idea how long the user will take to read the information, so another window is produced to tell the user that the information window will not go away until a mouse button is pressed. In this case, the mouse icon changes to show you that all three buttons are momentarily equivalent; pressing any of them no matter where you point is taken as a signal that you want to continue with P.

4.3 The Function Menu

Once P comes up, it displays the options menu from which you select a function by pointing the mouse at one and pressing **any** button. The actual menu is only the two rows of functions presented in solid-boarder boxes. For more menu or function items, see the section on "Pop-Up Menus" further on. If you ever get into a situation where you see overlapping menus, as when you invoke a pop-up function when the cursor is in the region of the main screen menu, it is the menu on top that is believed.

Below the main function menu you will see the name of the file that P is currently reading. Normally, the phrase shown is "File:" followed by the name of the file you invoked P with. For the case where you gave several file names when you invoked P, the phrase is "Current File:" followed by the name of the **one** you are currently reading, specifically to call your attention to this fact. If P is currently reading the standard input, the string "-- std input --" is seen in the file name field; indeed, P uses this as an internal name to signal that it is

reading standard input.

4.3.1 Where You Are At In The File Stream

Below the function menu you will see a box that is meant to show you how far thru the text you are currently positioned. This is a simple display using graphics which shows, via a grey-scale background, a particular percentage between 0 and 100. Actually, the display is always in one of two states - showing you actual an percentage, or it is a box filled with the message "End-of-File Not Yet Reached". The latter message is telling you why a percentage figure cannot be reported. Since P never reads from any file until it has to, it is impossible to say that what you see on the screen is X percent of the way thru the file until the end of the current file has been reached. (We could base all this on character counts instead of line counts, since you can find out the size of a file without reading it. However, what we are measuring would be less intutitive based on character counts, I believe).

There are several ways that P can reach the end of file. For starters, if you give the -e command line option, or if you invoke P to read from standard input, the entire file is read into memory automatically. Otherwise, the BOTTOM function is used when you want to specifically trigger this situation. Failing a SEARCH is another way to do it. Likewise, you can use any of the motion commands to "legitimately" arrive at the end of file.

No matter how it is done, P will start displaying the bar-chart percentage "icon" as soon as the number of text lines in the file(s) is known. The percentage figure is based on the middle line of text shown on the screen, in conjunction with the total number of lines accumulated in the file stream. Note that this includes all of the text lines accumulated thus far; P never discards lines of text that it has read once, even when these lines come from multiple files.

Once the bar chart appears, you will notice that it stays around until the next NEXT FILE is done, at which point the display normally reverts to the initial situation. This is again due to the fact that the size of the file stream is unknown because the file that is feeding it hasn't been completely read yet.

4.4 The Basic Commands

We present the P commands in the following sections, where they are grouped functionally according to the class of operation performed by each command.

4.4.1 The **motion** Functions:

Central to the user model of P is that once the user starts to go thru the text in a particular way, he will probably want to continue going thru it that way for at least a while. Therefore, the middle mouse button is best termed the **motion** key because it is this button that adapts itself to various types of motion thru the text.

For example, once you issue a PAGE function, you can get either forward or

backwards by PAGEs by simply pushing the middle button. Once you do a SEARCH, the middle button is then associated with reSEARCH, and you can skip thru the text seeing all the places where this string is found by simply pushing the middle button. If SCROLL motion is more what you prefer, you will see that a subsequent SCROLL again changes the middle mouse button to do this for you. In this way one is no longer forced to stay in the region of the menu - it suffices to point anywhere and simply use the **motion** button.

The following menu options are types of **motion**:

PAGE -This advances you in the current direction one page or screen-full, the number of text lines being dependent upon the current font. A screen redraw is always done with PAGE, the model being that this is the "flip flip" way that some people like to read documents. After PAGE, the first line at the top of the screen is the first line that wasn't visible before the PAGE.

SCROLL -Is like PAGE in that it advances you N (text) lines in the current direction, but a bitmap raster-op is used to move 80% of the screen in one lightening-fast operation. The N lines needed to fill the vacated part of the screen are subsequently read from the file or taken from memory, depending on whether you have already passed over this place in the file before. The current default for N is 10; it is displayed in the SIZE menu box since you use the SIZE function to change this default.

SEARCH -Lets you scan in the current direction for a string which you type into a prompt window. If you respond with <cr>, the last-used search argument is used. This is exactly equivalent to reSEARCH. After a SEARCH or reSEARCH, the found line is highlighted and positioned on the 10th line of the screen. This lets you see the surrounding context of what you found. It is somewhat lamentable that a screen redraw is ALWAYS done here; I may change this.

Whenever a successful SEARCH has been executed, an automatic LABEL command is issued by P behind the scenes so that you can get back to this place in the text if you want to. The label is the string "Found:" concatenated with the upcased search argument that got you there. Thus, even if you don't remember what you searched for or where it put you, you can still get back there: Use SHOW LABELs and you'll see such a label for each different SEARCH argument that you have successfully used.

If the last-found line is currently on the screen, a subsequent SEARCH or reSEARCH will start on the following line, if you are going forwards, or on the previous line, going backwards. That is to say: a double match of a string on a line is only found once. If you havn't done a search yet, or if the line you found is no longer on the screen, a subsequent search always starts with the top line (if going forward) or with the bottom line (if backwards) that **is** visible on the screen.

reSEARCH -Once you have already done a SEARCH once (which includes the **-s** option on the command line), simply pressing reSEARCH will advance you to the next occurrence of this string. The direction used in either SEARCH or reSEARCH is always the one that is currently in effect. Thus you can fail a SEARCH, change the direction, and then reSEARCH in the other direction without respecifying the operand.

Both SEARCH and reSEARCH do a case-**in**sensitive search. That is to say, both the source and the destination strings are mapped to upper case before the comparisson is done. This is why you see your search argument displayed in upper case in the "not found" message. I am in favor of adding a pop-up menu function to let you say that you want **exact**-match searches. I tend to think this function should establish a mode, like the verbose / terse model; otherwise we end up with 2 SEARCH commands and (presumably) 2 reSEARCH commands.

See SEARCH for a discussion of the automatic LABEL function that corresponds equally to SEARCH or reSEARCH. Note that since all labels in the P stream must be mutually exclusive, doing several reSEARCHes has the effect of moving this hidden LABEL along for you. There is no LABEL way to return to the Nth-previous place you were at.

AUTO -Perhaps one of the most controversial P functions, AUTO puts you into a mode where a "smooth scroll" is simulated to the best that the hardware can do. (See "How It Works", below). When you are in "Auto mode", the mouse buttons change meaning so the mouse icon on the screen is redrawn before and after AUTO. The **middle** button is how you exit auto mode and return to main screen menu mode.

Displayed in the "Auto xxx" menu window is a decimal number which shows you the "speed" that raster lines are scrolled. (For what it actually shows, see "How It Works", below). If you hit the left mouse button, this number is decremented by one and the new "speed" is displayed. Hitting the right-hand button lets you "speed up". Once enough of the screen has been moved for at least one line to be displayed, the line(s) is/are written to either the top or the bottom of the screen.

The AUTO function is not strictly a **motion** function because the middle mouse button does not link to it in the way that it does for the others presented above. Instead, AUTO is more like its own "sub system"; once you're "in" AUTO, you stay there until you hit a mouse button to exit that mode.

Because of the side effects having to do with direction or motion, the commands NEXT FILE, TOP, and BOTTOM could be considered motion-related.

4.4.2 Commands For **FILE Manipulation:**

The P model for letting you use several files at once is centered around the notion of a "P file stream". This "file stream" is what P accumulates as the user goes about perusing the text via the commands described in this document. The "file stream" is best described with reference to the commands that affect it, but see also the section "Where You Are At In The File Stream", above.

First of all, the file stream starts off with the N lines of text needed to fill the first screen. When reading from standard input, or if the user gives the -e command line option, the N used here is the total number of lines in the file. Also, if you start off searching (-s), P may skip some lines before the initial screen is displayed. In any case, a golden rule for P is that it **never** discards lines of text that it has had to read. That is to say, if P has skipped over lines of text for any reason, they are always remembered in program memory so that they can be redisplayed with ease. If an end of file is encountered as P reads to add lines to its file stream, the file is closed and the stream is considered to end there as far as subsequent motion thru this text is concerned.

The only other way to change the file stream is to use the NEXT FILE command. The thing to remember about that is that no matter where you are in the file stream - no matter what you have looked at and no matter what you **are** looking at, P always repositions internally to the end of the stream that has been read thus far, closes the current file if it is open, and then begins to add text onto the end of this stream. If the previous file **was** open, it must be because no user activity has forced P to read to the end of that file. In this case, and only in this case, P skips over whatever text may remain in this file **without** adding anything to the internal P file stream. After this point the new file becomes the "current file" and is reported in the "Current File:" window even when you are no longer looking at text from this file.

Just now there is no way to change the file names that you invoke P with. Likewise you cannot "re open" a file that you have already visited and closed. Since any text that you have gone by always remains in P's memory, it suffices to simply return to this place in the file stream to see the text. To do this, see the discussion below where we talk about the connection with the LABEL function.

The following functions provide the facility of manipulating files:

NEXT FILE -Is how you advance from one file to another. P sets the current direction to **forward**, repositions you to the end of the accumulated in-core file stream, (but doesn't do a refresh), closes the current file if it was open, (if you havn't visited all of the file yet), and fills the screen with the last line read from the previous file followed by the first PAGE of the "next" file. Of course, it there was no such "next" file, a message is produced and none of the above is done. The file names must have been given on the command line when you invoked P, and are the ones displayed via SHOW FILEs. In VERBOSE mode a message is issued telling you that the text of the indicated file is being added onto the end of the text stream that P is accumulating.

Each time you advance to a new file, the LABEL function is invoked behind the sceens with the name of the file as the label. Thus, after a NEXT FILE to, say, "p.doc", you can always get back to the beginning of this place in the P stream by using LABEL in conjunction with "p.doc". Just now there is no automatic LABEL for the first file; use TOP to reposition to there.

SHOW FILEs -Causes P to produce a display containing the names of the files you invoked P with, along with an indication of which file is currently being visited. Just now there is no way to randomly reposition yourself with respect to this list. Should there be?

SAVE PAGE -Causes P to write out each line that is visible on the screen into a file whose name is sought via the usual prompt pop-up window. If you respond <cr> to this prompt, the function is aborted; that is, you **must** supply a file name each time, and exactly this name is used. In VERBOSE mode a message is issued reporting on the file name and the number of lines written out. There is no way to add text onto a file; a create is always done.

SAVE TEXT -Is the equivalent of SAVE PAGE except that the entire P file stream is written out. Note that this is always the stream as it is when the function is invoked. Thus files or portions of files that have not been visited are skipped. In VERBOSE mode a message is issued reporting on the file name that was created and the number of lines written out.

Note that for both SAVE functions the text that is written out is completely independent of the current font. It is not the font characters that are written out; rather the internal ASCII codes (actually, binary bytes, since P can be used to display any type of file whatsoever) are output, just as they were input.

Admittedly, P's commands for manipulating files are not very extensive. Since P is an "afficher" of text and not primarily a manipulator of text, this seems to be fitting with the original conception of what P was meant to be. Now however it is not clear that P should remain this limited. How should the file model be extended? Should we add a few primitives to provide some more flexibility, or should the model be compltely revamped?

4.4.3 The **direction-change** Functions:

All relative movement thru the text uses an implied direction, be it forwards or backwards, and you change this direction by selecting the non-highlighted box at the right-hand side of the menu on the top row. The name of this box changes, but its position is always the same.

On the left-hand side of the menu you will see either "forward:" or "backward:" which **is** highlighted. This is meant to show you that direction is a prefix, effectively, to the other options shown. For example, when you see "backward:" highlighted and select "search", you are selecting "backward search".

The following functions have to do with direction changes:

BACKWARD -This function changes your current direction to backwards. To show this, the menu changes and "backward:" is highlighted and henceforth appears on the left-hand side of the menu. This is meant to show you that the direction is a prefix or modifier for the other options shown.

FORWARD -Normally you are already going forward, so "forward:" is highlighted and appears on the left-hand side of the menu. When you select the opposite direction, the two menu boxes are switched and "backward" becomes highlighted. The ':' with the function name is meant to imply that this is NOT a function - it is a way of showing you what direction is current. Indeed, if you select the highlighted direction "function" on the left-hand side of the screen, P will tell you that you are already going in that direction, just as it is redundant to select **any** highlighted function from a menu.

Because of the change of direction implied by the TOP or BOTTOM functions described below, they too could be considered in this catagory. A similar comment applies to the NEXT FILE command, described above.

4.4.4 The **Font Change** Functions:

Normally, a user of P has no need to be aware of fonts at all; P has a built-in default font that is used for the menus, user messages, and (by default) to display the text in the file stream. For doing font manipulations, however, the following commands exist on the various menus:

SET FONT -This command lets you change the character set that P is using to display the user text. (The menu and error messages are always output with a built-in font). If the font can be located and loaded, a screen refresh is done using the new font. Otherwise an error message is produced and the font that was in use before is not changed.

I have no plans to provide for multiple-font availability in P simultaneously; this is better left for a real Numelec <--> UNIX interface. Once you load a font, all others are "forgotten". The SET FONT command lets you do the equivalent of using a **-f** command line option in the first place. It is also closely related with the ZOOM function, which provides a more abstract or "structured" type of font switching.

ZOOM -Lets you go from one font to another, providing for the high-level notion of "zoom" (moving **in**) and "wide angle" (standing **back**). ZOOM works with a predefined list of fonts; each time you invoke ZOOM the next font from the list is selected for an implicit SET FONT command. Initially "the next font" means the first one in the list, and when you reach the last one a subsequent ZOOM returns you to the first one again. In verbose mode a pop-up window message is issued when ZOOM "circles around".

For ZOOM to even appear on the function menu, you must have previously defined the shell variable **P_Zoom_List** to contain a sequence of font file names, as in:

```
$ P_Zoom_List='font1.ft font2.ft smaller.ft /usr/lib/tiny.ft'
```

The idea is that a simple ZOOM function can suffice to let the user "go along" the sequence of predefined fonts. This is especially useful when you want to provide something specific vis a vis fonts, and you want to hide from the user the names, etc, of the fonts.

For details on the mapping between font file "names" and their actual pathnames, "Font File Search Rules", below.

SHOW FONT -Causes a pop-up window message to appear which reports the full name of the currently loaded font. Just now there is no way to have P report the elements of the ZOOM list. Should there be? If you really want to know, use the escape-to-SHELL function and look at the P_Zoom_List variable described above.

DEFAULT FONT -When you use P without specifying anything about fonts, a built-in or "default font" is used. This font is actually part of the program since it has to be there to write menus and error messages, etc. Thus, as far as the user is concerned, this font does not have a name and therefore cannot be selected with the SET FONT command.

To allow users to return to using the default font, the pop-up function DEFAULT FONT does exactly that. In VERBOSE mode you get a confirmational message. No matter what the mode, if you are already using the default font a message to this effect is produced and no font switching occurs.

Note that there are also some font-related functions available on the P command line. One of them is the equivalent of SET FONT, the other is related to the defaulting that ZOOM does. See also the "Font File Search Rules" section, below.

Should P let you use multiple fonts at the same time? Currently there is no way to do this, mainly because I'm unwilling to invent yet another "standard" for specifying this. Does anyone think that backslash-f followed by a single digit number (representing - somehow - a loaded font) is anything like a standard that we could adopt, here?

4.4.5 The **Jump To** Functions:

Aside from relative motion thru the text, the "Jump To" functions provide a way to move to a specific place, be it the bottom, the top, or to a place that you have already seen before and have put a LABEL on.

TOP -Repositions you to the top of the first file you have read. Even if you have NEXT FILEd your way thru several files, TOP will get you back to the very beginning. (But see LABEL and GoTo with respect to NEXT FILE). After a TOP function, the direction is always set to forwards.

BOTTOM -Puts you at the end of the CURRENT file. This guarantees that all of this file will be read into memory, and subsequent operations within this file will NOT cause any I/O to happen. Currently there is no way to "bottom" your way to the end of a **list** of files. After a BOTTOM function, the current direction is always set to backwards.

TOP and BOTTOM are the only functions that have the notion of automatically adjusting the direction because you have "arrived at the end". If you PAGE your way to either end of the file stream, you must manually change the direction before PAGE (or any motion command) will take you any further.

LABEL -Prompts you for a string, and henceforth associates this string with the screen positioned exactly as it is when you do LABEL. The model here is FINGERpoints or "book marks" in the file(s). Any character string whatsoever is allowed, including <null> - but watch out: GoTo does an exact match and is not very forgiving.

P allows you to have at most one screen LABELed with any given name. Thus, each time you invoke LABEL, a check is made to see if establishing this LABEL would duplicate an existing one. Whenever this is the case, P simply goes ahead and destroys the old LABEL; in VERBOSE mode a message is issued to this effect. When LABEL is called internally by P (in conjunction with SEARCH, reSEARCH, or NEXT FILE), no such message is issued no matter what the mode.

SHOW LABLEs -Provides a way to find out which LABELs are currently established, either by the user directly, or by P indirectly as side effects of the SEARCH, reSEARCH, or NEXT FILE commands.

GoTo -Lets you reposition the screen to the given LABEL. You can have the <null> string as a label, but like any label, you can only have one with the same name. If you reuse a label, the first screen that you used it for is no longer relevant. In VERBOSE mode a message to this effect is produced when you reuse a label.

Previous to this version of P, you could give "?" as the operand to GoTo and a display of all existing LABELs and their associated line numbers would be produced. This "feature" has been removed and you can now give literally any label string whatsoever. To see what your labels are, use the SHOW LABELs command described above.

Here's another idea for a motion type function: NEXT LABEL. This function would take you in the current direction to the next LABEL that you have set. If you want to continue on after that, again doing NEXT LABEL would place you at

the next screen-full that you have LABELed. The set of LABELs could be considered circular, much like the zoom list. This function would make it practical to use LABELs for a particular marking of your source stream, and to then skip from one "interesting" screen-full to another without having to sort out what all the names you made up are. Comments?

4.4.6 Some **Miscellaneous** Commands:

All of the other functions available in P are described in this section. They permit you to scroll horizontally, to issue an arbitrary command to the UNIX shell, and other "option" and "set" type functions.

Refresh -Does a clear and then a redraw of the screen.

LEFT -Returns you to the standard mode - where the left-most column of the text is displayed at the left of the Numelec screen. This menu box is usually highlighted, because it is usually the case that you are viewing a left-column-offset file stream. Any time that this is not the case, LEFT is no longer highlighted.

RIGHT -Provides a way to view files that can be much wider than one screen. You are prompted for an integral number, N, which is then used as the column offset (from the left). For example, if you give "10<cr>" in response to this prompt, after the redraw you will see your file offset by 10 characters. A subsequent RIGHT function is cumulative. i.e. the model is that you want to go right in a step-wise fashion. You may enter negative numbers here to move back towards the left, but see LEFT for a way to easily return to the "standard" position. If you enter the <null> response to the RIGHT prompt, the default column offset is 1.

Once you are to the RIGHT of LEFT, the number of columns that the text is offset is written into the RIGHT menu box. When this number is 0, no column-offset number is reported in the box.

!<command> -Lets you enter any UNIX command, which gets executed in a child process. Just now the standard output of the contolling terminal (the TTY) is inherited by the child - NOT the Numelec output stream. Undoubtedly there is lots of cleanup to do in this area. When you return to P, a screen refresh is done if the program that you ran actually changed the bitmap screen.

SIZE -Lets you specify the number of **text** lines that are skipped each time you invoke the SCROLL function. You are prompted for an integer N, and as long as you give 0 < N < max, this becomes the default for SCROLL. This sort of option should probably not be on the main function menu, but it is there now because it's the only way (short of a SHOW SIZE) to show users the current setting of the variable. The current default is displayed in the SIZE menu box.

4.7.2 The AUTO-Scroll Function

For "smooth" reading on the screen, the AUTO function exists to let you simply "sit back and watch". Really, though, this function was invented to demo the "move-screen Raster-OPs". Hence the speed-up and slow-down mouse buttons.

To explain how this works, let's say that there are SCR raster lines of text on the visible part of the screen. This does not include the menu. To make a scroll happen in software, you need to move a certain part of the screen up or down, zero out the remaining part of the screen, and fill this blank space with text. In P, the number of lines that are moved is the same number that "Auto xxx" prints in the menu box. Note that these are raster lines, not lines of text, though. Let's call this MOV. The auto function is a continuous loop, then, which moves (SCR - MOV) lines either upward on the screen, or downward on the screen, depending on the direction of scroll. In this loop, I move "most" of the screen, zero out the MOV lines, and tally up how many raster lines I have zeroed so far. Once I have looped enough to have enough raster lines to print the next line(s) of text, it is read from memory or the file and displayed on the screen.

Currently, if you set the "speed" to 19, you see exactly one line of text for each screen-move raster-OP that gets exectued. This is because the built-in font requires 19 points to accomodate text lines plus the surrounding spaces. If you set the speed to more than 19, this function begins to act more like SCROLL except that AUTO is continuous. For speeds greater than 19 raster lines, more than one line of text is sometimes written on the zeroed-out space each time the rest is "rasterED" up or down.

The Numelec "scroll register" is not used in any way to effect this or any other kind of scroll. This is primarily because (1) there are menus, etc, on the screen and we don't want to scroll that, and (2) we don't want to preclude the possibility of storing fonts or other information in the "hidden" part of numelec memory, so we can't simply write all over the 2048 raster lines that there are.

If you have any idea of how this scrolling hardware CAN be used to scroll within such constraints, please let us know about it.

4.8 Further Information

For more information on using P, please contact the author. Since P was written explicitly to demonstrate the usage of the Numelec, the mouse, and the raster-OPs, we will provide source listings of P to anyone wanting to develop bitmap software for Smx and the Numelec. For usage information on P explicitly, you can get a printed version of this text by simply copying the ASCII file:

/numelec/Numelec.Doc/p.doc

if it was put there on your system (there may be a leading prefix pathname); otherwise ask the person who installed P or contact the author. Although you can print this file on your line printer, note that it is not paginated; this makes

it much better for looking at it on-line with P.

Any suggestions on how to improve P or this text would be appreciated.

Contact: Kevin Pammett, INRIA, Rocquencourt. (3) 954-9020 poste 3160 or 3442.

[end of P tutorial text -- updated as of 21 March, 1984]

VERBOSE -Lets you change the verbose/terse mode back and forth, just as in the font editor, FE, and other tools. This is exactly the same option that you can set via the command line **-v** and **-t** options. The reason that VERBOSE is not on the pop-up menu is that function on the pop-up cannot double as indicators to tell you the current setting of something. Indeed, all menu items that involve highlighting or adding numbers or parameters to the display text are on the main function menu for exactly that reason. Is this worth it? Instead, should P get into highlight (etc!) menu items even in the pop-up menu?

QUIT -Is the standard way to exit P. Your terminal characteristics are re-established, and a success exit to UNIX is done. This is exactly the same function that is normally associated with the mouse button labeled "Quit P".

4.5 The Pop-Up Menu

For ergnomic reasons, all functions available in P are not always displayed in a menu which is always on the screen. Instead, only the commonly-used ones are always visible, and a pop-up menu is provided to make the others available.

The left-hand mouse button is normally labeled "Select". If you press this button in the region of the menu, this is exactly what it means. However, if you press SELECT when you are not pointing to a menu, a pop-up menu of functions is produced in the vicinity of where you were pointing. After this point, any mouse button can be used to select from the pop-up menu, and if you SELECT anywhere outside the menu, this is taken to mean that you don't want to select a pop-up menu function. No matter what you select to do, having done so, the menu disappears and the function is invoked.

Some functions appear only on the pop-up menu, but are still described in their functional sections in this document. For consistency sake, there is one function specifically associated with the pop-up menu:

Cancel Pop-Up -If you actually select CANCEL POP-UP, the pop-up menu disappears and you return to P just as if you had not invoked a pop-up function. There is no difference between selecting this function and moving the mouse outside of the menu and then pressing any mouse button.

The pop-up menu facility in P has been designed as part of the SM90 bitmap tool set, and will be available as part of the standard "menu package". If you have ideas on what should be provided in this package, please let us know.

4.6 Font File Search Rules

By design, when any reference is made to a font, it is not usually the case that the actual name of the file that the font is stored in is given. This frees up users from having to know the actual location of font files, the fact that these file names end with ".ft", and etc.

In order to present this issue of "font file search rules" consistently, I have written about it in the document that describes the whole tool set available from INRIA that work with the Numelec bitmap. For P's usage of fonts, suffice it to say that P follows the rules described in this other document.

To find out the name of the font that is being used by P, see the SHOW FONT command. To return to using the built-in font that P uses for its messages and menus, see the DEFAULT FONT pop-up command.

4.7 How It Works

Since P is really an experiment in evaluating the Numelec hardware, some words on how it all works seem appropriate...

4.7.1 File Motion

P keeps all of the text that it has read in main memory. This permits backwards motion and search, LABELs and GoTo, and a number of other things. It also lets us really see the speed of certain raster-op functions without the overhead of doing any I/O. Unfortunately, it also puts a limit on how much text you can look at; currently this limit is 8192 text lines. There is also a limit on the character width of any line - 255 characters, and you cannot invoke P with more than 50 file arguments. The actual width of a line on the screen changes when you specify that you want P to use a different font. Another limit is that you cannot put more than 15 font names in the P zoom-list variable.

Only those lines that have to be read are read and remembered. For example, if you have only gone part way thru a file, and then do NEXT FILE, the original file is closed, and the next file starts immediately after the last line that happened to have been read from the first one. After this, there is no way to get back to the original file and read the text that you skipped the first time. Since all text that has been remembered is always available, though, you CAN easily review text that you have seen (or searched over) once. It's not terribly clear to me that this is the desired behavior of P, but I don't want to impose the overhead of having to read to the end of the current file when the user has clearly said that he wants to see the NEXT FILE... Comments?

CHAPTER 5

A Bitmap Interface For The UNIX SHELL

In Retrospect: This is the most important piece of real **research** into human interfaces that I did at INRIA. Several new areas are explored, including: a menu which is completely set up by the user of a tool, the ability to dynamically edit and deposit strings into a menu, a highly-specialized mouse-driven string editor (for command strings), and a (command) HISTORY mechanism that is accessed via an adaptable menu package.

I regret that this work was done very late during my stage at INRIA. I would very much have liked to have been around for the first 6 months of usage for the bitmap SHELL. After all, it is the humans that **use** human interfaces that are the final word on whether they're effective or not.

The rest of this chapter contains the BSH document verbatim.

BSH – A Bitmap Interface For The UNIX SHELL

(Tutorial)

Kevin Pammett

I.N.R.I.A.
Rocquencourt, France.

5.1 SYNOPSIS

bsh [-v] [-t] ...

BSH is a new version of the Bourne shell, otherwise known as SH. A menu-driven, "bitmap" interface has been prototyped and linked directly with a slightly-modified version of SH. The result is a shell that you can use just like SH, except that there is a menu/mouse mode, a HISTORY mechanism, editing of the current as well as "all" gone-by commands, and many other improvements in the human interface. This paper, plus the companion paper on the pop-up editor, IED, collectively describe using the bitmap shell interface.

Note: in this document I use the nomenclature **BSH** *to refer to the bitmap shell. On the UNIX system distributed by INRIA, the actual name that you type*

is 'bsh' (in lower case).

This version reflects BSH at "baselevel 1", which coincides with version 3.2 of Smx. This version of this document supercedes the draft versions that were made available to some of you. Since a "bitmap interface to the shell" is somewhat of a research problem, this is a "working design document" for BSH; the text contains numerous questions intended to further the development of the interface via user feedback.

5.2 DESCRIPTION

BSH is a "bitmap shell" for UNIX (Smx) for those who have a Numelec bitmap terminal. Actually, I refer to it as a "SHELL interface" since the commands that you can execute via this shell are exactly the same commands that the Bourne SHELL provides. BSH is literally an interface linked in with the SHELL.

You can use BSH as a login shell, directly, or you can invoke it as a command via "$ bsh", depending on whether you want to go in and out of the SHELL interface during a single login session.

While we are waiting for the Smx kernel to provide multiplexed I/O for the keyboard and the mouse, you will see that there are two "principle modes" that you are in when using BSH, and that the modes are mutually exclusive. If you elect to invoke commands, etc, via the mouse, you are in "mouse mode". On the other hand, you can go into "TTY mode" and type your commands just as you would using SH. These modes are described in more detail, below.

The principle advantages of BSH over SH are:

- -No matter what mode you are in, the date and time and the current directory are always visible and in the same place on the screen; they are kept up to date "automatically" by BSH (every time you hit <cr>, whether you execute a command or not).

- -Even in "TTY mode", where you type in commands just as for SH, you can elect to **EDIT the command line** before it gets executed.

- -There is a HISTORY facility which lets you recall previously executed commands for the purpose of editing them, executing them, storing them in your on-screen menu, or all of the above.

- -There is a personalized menu on the screen in which you can edit, pick-up, deposit, and execute the items. I refer to this as the STRINGs menu although you might think of it as the "Personalized Shell Commands" menu. You can set up this menu in your **.profile** command file.

5.3 Presentation

Before we go into more detail on specific BSH functions, we present the screen layout that you see when you invoke BSH. Also described are some other aspects of "presentation".

5.3.1 The Function Menu

Once BSH comes up, it displays what appears to be the "main menu" at the bottom of the screen. In fact, there are actually two menus there. The bottom row of function boxes contains the FUNCTIONs that are predefined by BSH; they all have to do with manipulating the environment, going in and out of "menu mode", and etc. The font used in these boxes is larger, and boxes themselves are larger. Since this is considered a "real menu" (like all other bitmap tools), you select a function from this menu by pointing the mouse at a menu item and pressing **any** button.

The second part of this menu is what I refer to as the "STRINGs menu". (I would like to use the term "COMMANDs menu" since the user puts SHELL commands in this menu, but the term **commands** applies equally well to built-in BSH commands and SHELL commands). The term STRINGs promotes the notion that these are simply character sequences that are manipulated via IED and BSH; they only become commands after they leave the interface and are presented to the real SHELL for execution.

Physically, the STRINGs menu is the top two rows of the main-screen menu. These boxes are smaller than the FUNCTIONs menu, and the boxes **will** be variable-shapped - all to help you distinguish this from the "canned FUNCTIONs" menu. A smaller character set is used to display these boxes, for the same reason. This layout is described in more detail in its own section below.

You can select one of the BSH intrinsic functions from either the FUNCTIONs menu, or from the pop-up menu (described near the end of this document), by pointing to a menu item and hitting **any** of the three mouse buttons. Within these menus, all three mouse buttons are considered equivalent; outside of a menu they are almost always distinguished. In the STRINGs menu, too, they are distinguished. In effect, the STRINGs menu is a breadboard (prototype) for something beyond a menu as we know it. See below where we describe the STRINGs menu in detail.

5.3.2 The Information Window

Below the on-screen FUNCTIONs and STRINGs menus you will see a small region that I call the "Information Window". Here BSH displays information that is (supposedly) relevant to the user at all times. This is distinctly different from its pop-up counterpart described in the section, "The Mouse Icon", below.

Just now the only things that appear in this window are (1) the name of the current directory and (2) the date and time. Eventually this region might be used to display other things or even to converse with the user.

The information window is the property of BSH. It may be obsconded, if you invoke a tool that simply takes away the whole screen (like P or FE), but immediately upon return to BSH the window will be reconfiscated.

BSH keeps both the current directory and the time of day up to date automatically. The former is recalculated after each time you do CD or CHDIR; the latter is calculated each time the SHELL executes a command of any sort, including when you type only <cr>. What else could we use this region for?

5.3.3 The STRINGs Menu

The STRINGs menu is best thought of as a "normal menu with special goodies". With a standard menu, you as a user can't change what is **in** a menu, and the menu items are all the same size. To accomodate this, the designer of a bitmap application takes care to make the menu item strings all approximately the same length. For any given program menu, then, the order of items, font used, and presentation in general, are all fixed when you build the application.

The STRINGs menu is a prototype that goes beyond this. The user gets to define, tailor, and even dynamically edit and deposit into this menu, as we will describe in detail below. Just now the menu item boxes are all the same size, but this will (hopefully) be fixed in the next version. It is very definitely the intent that the menu item boxes be calculated depending on the size of the actual strings offered in the menu. (For the time being, our menu package is too rigid for this).

Just now there are three ways that (command) strings end up in the STRINGs menu. (1) a few of them are pre-programmed in, (2) the user puts command strings there dynamically with the DEPOSIT function, and (3) the user, via the MENU command described below, specifies command strings that he wants to appear "automatically" in this menu. In actual fact you can **edit** a menu entry, too, but this is considered a special case of (2), above.

The purpose of the STRINGs menu is essentially that it provides a mechanism for the user to offer himself an easy choice of whole or partial SHELL commands for execution. Since you can point to an item in this menu and press the **Execute** button to have it executed, one will normally store whole or partial SHELL commands here. In actuality, however, there is nothing that forces the STRINGs menu items to be SHELL commands. In fact, they are simply character strings that you can put in the menu for any reason - like to remind you of the

name of something you're working with, etc.

The intent is that **none** of the STRINGs menu is predetermined by BSH. Just now the case of (1), above, is a counter example to this; I offer predetermined commands for now only to show people the kinds of things they might like to put there. In the future this part of the menu will probably be empty by default.

It is useful to think of the STRINGs menu as somewhat like the HISTORY menu but more tailorable, much more easily accessed, and something that is visible all the time. The connection between the two is that you can use the **Pick-Up** function to specify a command from your own HISTORY and literally DEPOSIT it into your STRINGs menu. This is what you should do if you find yourself going to the HISTORY sub-system a lot to pick up a particular command; if you put it on your screen, you will be able to select or edit it with ease. Likewise, if you point to the execution window and do **Pick-Up**, you can recoup the last command executed and DEPOSIT it into your STRINGs menu. This is a conceptual simplification of its HISTORY counterpart. It is also useful to **Edit** the items in the STRINGs menu, or, likewise, to **Edit** the last command executed.

Of course, putting all of the fancy goodies aside, we should stress that the expected usage of the STRINGs menu is in conjunction with the **Execute** function. The whole point of the bitmap shell interface is that you can simply point to UNIX commands that you want to execute, and have them executed directly. See the example below for a way to do exactly that with an automatic option to edit the command first, each time that it is executed.

5.3.3.1 The MENU Command

The only command that I have actually added to the SHELL is MENU. If you type "$ menu" with no operand, you will go from TTY mode to "menu mode". See below for the two sections where we detail these modes. In any case, this functionality will be rendered obsolete when/if the Smx kernel provides applications with the ability to wait for a mouse click **or** a character being typed on the keyboard, which ever happens first. Just now this is not possible; hence the two modes.

The other use of MENU is analogous to EXPORT and READONLY; all three are SHELL commands that take one or more names of **variables** as their arguments and alter the attributes of the indicated symbols. EXPORT makes a symbol be a candidate for being passed on to a child process. MENU says to BSH that the contents of the indicated symbol(s) are to appear in the STRINGs menu.

For example, the sequence:

```
$ look_for='grep  ^  [a-z]*.c'
$ menu look_for
```

creates the ordinary SH variable `look_for' with the (partial) grep command as its value. The second part of the above says that the grep command should appear in the STRINGs menu. Each time you execute the MENU command BSH rebuilds this menu according to the rules described in the section below.

Note the ^ character in the above grep command. It takes the place of the first operand to grep, which must always be supplied before the command can be successfully executed. On the screen this "edit character" (type ^P or ^N, for "previous" or "next") appears as an up-pointing arrow; here we see only the "fleche" or "arrow" ASCII character since that is the best the typesetter can do. This is the same special character that is recognized when you enter command lines by typing them directly. It is taken to mean that you want to edit the command line before it is executed. When you store such a character in a STRINGs menu item, the effect is that you have a command that **automatically** implies editing before each time you execute it. For commands where you almost always want to supply an argument "on the fly", this fills the bill perfectly; you always get to fill in what you want, and you never have to delete the argument from the previous invocation.

5.3.3.2 Rebuilding The STRINGs Menu

When you invoke BSH, or when you execute a MENU command interactively, the interface goes thru a process of rebuilding the STRINGs menu. This will undoubtedly improve, but for now the interface tries:

(a) -to fit in all the predefined strings,

(b) -to retain everything that you have specifically DEPOSITed into this menu during this session,

(c) -to then fit in all the commands that come from symbols that have the attribute: MENU.

If BSH runs out of space in the menu during this process the overflow menu items simply do not get put into the menu. No message is issued to this effect, primarily because I invisage this whole aspect being rewritten to use a more dynamic menu package interface. The fact that the menu item boxes are all the same size is simply inacceptable in this situation.

Currently there is no way to **stop** BSH from offering the initial commands that it does in the STRINGs menu, and if you define a command with MENU it will **not** take the place of any of the predetermined commands. You can, however, reuse one of these menu boxes by editing it or by depositing another string into its place. If you edit-away the string, then this menu item will become as available as one that was never used in the first place.

There is no DELETE function associated with the STRINGs menu items. This is because one can do the equivalent of this by using **Pick-Up** with an empty menu box and then depositing that on top of the item you wanted to get rid of. This assumes that there is always an empty menu item around for this purpose. With the current version, where the boxes are all the same size, this is really not viable. However, once we have variable size menu item boxes this "problem" will go away. It is almost inconceivable that there will be a significant number of cases where the existing strings fit so exactly into the menu frame that there is no "hole" left. Furthermore, if the menu manager arranges to put this hole in the **center** of each row, we will have the aesthetic side effect of a "balanced" (visually) menu, too.

5.3.4 The Command Execution Window

All of the screen that is not described in the preceding sections is referred to as the "Command Execution Window". In practice, this is most of the screen. Intuitively, think of this part of the screen as the "execution space"; it is where commands are executed by the SHELL.

This space is a real window in the sense that the Smx kernel limites I/O to being in this region. That is, if you execute any ordinary UNIX process that does I/O via the terminal, only this part of the screen will be affected. See the BITMAP paper where we talk about scrolling regions for more details on how this happens.

If you execute a bitmap application that does its own I/O - like FE or P or even EMIN, you will see that they do not limit themselves to this scrolling region. We expect that this will change in the near future. In any case, BSH takes back the screen when you exit the child bitmap process.

As you will see from the mouse icon, the mouse buttons very definitely have a meaning if you point to the execution window and click one of them. Since this space represents "execution of commands", the **Pick-Up** and **Edit** functions, when pressed here, imply the last command you executed. For the **Execute** function, though, one sees a distinct difference since this button is "always" reserved for Pop-UP functions. For more details see "Mouse Mode", below.

5.3.5 The Mouse Icon

To the right of the function menu you will see an **icon** that is meant to tell you the meaning the mouse buttons. This is the same icon that is used in all other bitmap tools distributed by INRIA. Shown in the icon are three triangles meant to symbolize the three mouse buttons. Follow the dotted lines from these buttons to see which function names are currently associated with each button. During the operation of BSH the phrases attached to the buttons are likely to change radically.

The left-hand mouse button is normally called EXECUTE and is the one that we say you should use in a menu to invoke a function. If you hit this button outside of the main screen menu, another menu of functions will appear and you can select one of them as is described in the section "Pop-Up Menu Functions" near the end of this paper. Note that you must use the EXECUTE button, here. In IED pressing **any** button outside of the editing window will invoke a pop-up, but this is because there is no other meaning for the "extra" buttons here in IED.

If you make some sort of error while using BSH, a message is usually produced in a pop-up window. In general, this window is reverse videoed so that you will take notice of it, and a time delay is built-in to BSH so that the window will automatically go away without the user having to do anything. See the document on P for further comments on this issue.

Like IED, BSH has a mode where the mouse is **not** active. When this is the case,

a special hand-drawn "mouse icon" is displayed in place of the usual one. This icon tells you why the mouse is inactive (because you have selected TTY mode) and what you have to do to get back to mouse mode (type the parameterless BSH command: $ menu). The form of the icon is the same as usual, but there is no picture of the three mouse buttons. The icon is "hand done" to draw attention to the fact that this is not your usual mouse icon - it tells you that you can't use the mouse.

5.3.6 The Pop-Up Menu

For ergonomic reasons, all functions available in BSH are not always displayed in a menu which is always on the screen. Instead, only the commonly-used ones are always visible, and a pop-up menu is provided to make the others available.

The left-hand mouse button is normally labeled "EXECUTE". If you press this button in the region of the menu, this is exactly what it means. However, if you press EXECUTE when you are not pointing to a menu, a pop-up menu of functions is produced in the vicinity of where you were pointing. After this point, any mouse button can be used to select from the pop-up menu, and if you SELECT anywhere outside the menu, this is taken to mean that you don't want to select a pop-up menu function. No matter what you select to do, having done so, the menu disappears and the function is invoked.

Some functions appear only on the pop-up menu, and some are available from more than one menu. For those that are not described within one of the functional sections in this document, see "**Miscellaneous Commands**", below.

For consistency sake, there is one function specifically associated with the pop-up menu:

Cancel Pop-Up -If you actually select CANCEL POP-UP, the pop-up menu disappears and you return to BSH just as if you had not invoked a pop-up function. There is no difference between selecting this function and moving the mouse outside of the menu and then pressing any mouse button.

The pop-up menu facility in BSH has been designed as part of the SM90 bitmap tool set, and will be available as part of the standard "menu package". Especially when you see more than one application sharing a pop-up menu, as is the case here with BSH and IED, how this important real estate can be divided up, shared, and co-operated with is not evident. Likewise, even the HISTORY mechanism is a pop-up menu in disguise; we are not sure about this being "a standard" yet; I think more work needs to be done to see just how and in what direction we want to provide all the "user hooks" that I invented to use menus for what HISTORY is working from.

5.4 How To Use BSH

There are three major situations that you can be in when you use BSH. If you hand type commands to the bitmap shell, as you are probably use to doing with SH, see "TTY Mode", below, for the details. Note that if the on-screen menu item called "TTY Mode" is highlighted, then you are in this mode and no mouse movement will be noticed. If it is **not** highlighted, no amount of typing will do any good, even though the typing is (currently) echoed on your terminal. (This is considered type-ahead, and it will be executed when you enter TTY mode).

If you are not in "TTY Mode", then you are in mouse mode. Most of this paper details what you can do in mouse mode.

From either mode you can call up the one-line editor, IED. This editor is usually used to edit SHELL commands or menu items from the STRINGs menu. See the section below where we give an overview of using IED from BSH. There is also a companion paper that describes IED in more detail.

5.4.1 TTY Mode

So-called TTY Mode is meant for those who want to try out the "bitmap shell" but who don't want to change their current typing habits very much. Except for the displays on the numelec screen, typing and executing commands in "TTY Mode" should be the same as what users are familiar with already.

When you first invoke BSH, you will see that mouse icon doesn't really look like an icon. This is done on purpose to let you know that if you use the mouse at this point BSH will not take any notice of it: In tty mode, which this is, you simply cannot use the mouse at all. Indeed, there is no mouse cursor, for exactly that reason.

This mode is (should be!) 100% upwards compatible with the current SH on Smx. It is the default mode, but I'm not sure if we should leave it that way. When one calls up a "bitmap interface", presumably one should get into a mode where the mouse is usable... Comments?

There is one major reason that I say "upwards compatible" and not simply "compatible". In standard UNIX there are certain control characters which have a particular meaning when you type them as part of your input to the SHELL. The ^S and ^Q TTY commands are good examples, here. You can be in the middle of typing anything, but if you type ^S this stops the output going to the terminal. The ^S character is not considered part of the standard input; it is completely absent from the character sequence received by the program reading the input you typed.

BSH defines two more control characters, ^N and ^P, for next and previous. As commands all by themselves, they are taken to mean that you want to edit (in the IED sense) either the next (forwards, time-wise) or the previous commands from the HISTORY list. At the command input level there is really no notion of next, since by definition you are already typing the most recent command. Once you get into the middle of the history sequence, though, both directions

make sense.

If you have already typed some characters in your command line, the interpretation of ^N or ^P is slightly different; this is taken to mean that you want to edit the **current** command before it gets executed. It does not matter where in the command line the ^N or ^P character appears, or how many of them there are. They are all removed before you are brought up in IED editing the command string. Since the shell reads in normal tty mode, you must hit <cr> as usual before BSH will take notice of these control characters.

Once you get into IED editing any command line, see the pop-up functions that let you step forwards and backwards thru the HISTORY commands. See also the ^Xn and ^Xp (next and previous exit modes from IED) keyboard commands in IED.

As you will read on the "mouse icon" associated with this mode, you leave it by typing the BSH command "menu".

5.4.2 Mouse Mode

When you first invoke BSH in menu mode, you will see that the cursor always appears at a fixed place on the screen, and that it starts out flashing. This is specifically to tell you that BSH is ready for you to use the mouse and invoke commands. As soon as you touch the mouse the cursor stops flashing never to start again. This is to say, once you are in control, BSH lets you know this by **not** flashing the cursor.

In general, the cursor follows the mouse and its form changes depending on whether you are in the region of a menu or not. To let you know when you can SELECT from **any** menu, the form of the cursor is a right-pointing arrow. You will see this whether you are in the main functions menu, the associated pop-up menu, or even the HISTORY menu.

As soon as you move away from one of these menus, a bulls-eye cursor is displayed. The bulls-eye, rather than a question mark, is used to tell you that there **is** a meaning attached if you use the mouse buttons when you are in this area.

There are three functions available via the mouse:

EXECUTE -Invokes a built-in function when you point to the FUNCTIONs menu (the bottom row of the on-screen menu). If you point to the STRINGs menu, this function causes the indicated SHELL command to be executed. Off menu, this button invokes the pop-up menu, described below.

PICK-UP -When you use **Pick-Up** in the STRINGs menu, you can pick up a STRINGs command for the purpose of depositing it elsewhere in that menu. You are queried for the place you want to deposit the command. If you respond by pointing outside of the STRINGs menu, the **Pick-Up** function is aborted.

If you point off menu and do Pick-Up, you get to pick up the last command executed. Again, you are queried for where you want to deposit it.

EDIT -Lets you edit something from the STRINGs menu, or the last command executed, depending on where you point. If you EDIT off menu, the assumption is that the command should be EXECUTEd afterwards. On the other hand, EDITing a STRINGs function directly by pointng to it carries with it the assumption that it should be deposited if you leave the editor with **Done Edit**.

Mouse mode is not currently the default when you invoke BSH. Should it be? Also, how 'bout a command line option (-m) to cause mouse mode to be entered when you invoke BSH.

This mode is completely different from what you see when you use IED, even though IED is "in" BSH. Watch the mouse icon, especially when you go from BSH to IED and back again.

5.4.3 The Built-In Command Editor – IED

A companion document describes the command-line editor that you see in BSH. I call this very specialized editor IED, after "Icon EDitor", since what IED actually does is to provide a high-level way of manipulating and displaying a chain of icons. A character string is a simple case of a chain of icons.

Most of the details of using IED are described in the IED paper. There is **keyboard mode**, where you have a host of EMACS-like control characters for moving around, deleting characters, etc., and there is **mouse mode**, where you get to edit, albeit in a limited way, with the mouse. See the IED paper for all of these details. In this paper we give some of the info that ties the two together.

First of all, the reason that I talk about "using IED from BSH" is that I expect that some day many of the highly-interactive tools that are available with Smx will use this pop-up editor. Ideally the user learns one interface for doing this very simple (one-line) kind of edit operation, and nothing else has to be learned when a similar situation comes up with another tool.

Another example of where IED might appear is in the command iterator, ITER, also described in a companion paper. In ITER there is always a "template command" which is instantiated with the current element from a set of elements, just before the resulting command is executed. To provide users with a way to change this template command, a "Change" function is provided which calls up /bin/ed. Naturally, if you are ITERating on a bitmap terminal, it should be IED that is called up. This is another of those "wait for the next version" scenarios.

If you view IED as being part of BSH, then you only have to learn the commands described in the IED document and you will know how to use BSH. However, here is a slightly-different model - the one that this software architecture is actually based on.

5.4.3.1 A Tighly-Coupled Software Architecture

IED and BSH are examples of tools that co-operate in a tightly-coupled software tool architecture. One gets the impression that IED is simply a "goodie" that is offered as part of BSH, but, by design, this is not the case. Indeed, IED is linked-in with BSH - you know this because each has a fairly intimate idea of what the other is up to.

In fact, however, there is a very well-defined interface between the two. BSH calls up the editor passing on a font to use and a character string. The editor lets the user do whatever, and eventually returns to BSH with (a) a resulting string, and (b) a return code. The return code is normally TRUE of FALSE (i.e. 1 bit) and the caller simply takes the resultant string and does whatever it was going to do with the original string (on a TRUE return), or it abandones this operation, (on FALSE).

In the case where a **character** is returned as the return code, however, the caller (BSH, in this case) is allowed to take arbitrary action based on this character. From the user point of view this boils down to the ^X command, in which the user types a character following the ^X and this character is what is passed back by IED. The editor has "no idea" of the significance of this "return character", it simply passes it back to the caller and does its normal clean-up operation. If the caller decides that control should **return** to the editor, IED is **re**invoked with the changed string, even though it doesn't know the difference between this and having been called with this new string in the first place. This is a protection mechanism to help users who didn't mean to type ^X.

If you leave IED and return to BSH by typing ^X followed by one of the characters described below, BSH will take the associated action.

^Xm -For "Menu". The keyboard mode command ^Xm is equivalent to the "Execute & Deposit" pop-up function. No matter how you entered IED from BSH, this function is how you do both an "execute" of the command string, and a "deposit" of it into the menu. If you did not get the string in the first place from the menu, you will be prompted for where to deposit it.

^Xe -The keyboard mode command ^Xe is equivalent to the "Execute String" pop-up function. No matter how you entered IED from BSH, this function lets you send the edited string off to UNIX for execution.

^Xd -The keyboard mode command ^Xd is equivalent to "Deposit into Menu" pop-up function in IED. This function is useful when you entered IED by some BSH function other than "Edit Menu" and you later decide that you want to have the edited string deposited into the menu. Unless you did enter by "Edit Menu", you will be queried for where in the menu you want the string to be deposited.

^Z -The IED command ^Z actually returns a "z" return code to BSH and is therefore equivalent to the "DONE IED" pop-up function in IED. This is the standard way to exit IED. A success exit is done to the calling procedure, and it takes whatever action it normally does with the edited string produced by IED.

^Xq -The return code "q" is equivalent to the "QUIT IED" pop-up function. This is the **non**-standard way to exit IED. BSH simply ignores the operation that caused IED to be called in the first place.

^Xp -"Previous" or "**Step Backwards** thru HISTORY". You leave IED and enter again, this time editing the previous (History) command. If you are already at the beginning of the HISTORY list, a messge will be issued and you will reenter IED with the string you had before.

^Xn -Next (forward) or "**Step forwards** thru HISTORY". You leave IED and enter again, this time editing the next [more recent] (History) command. If you are already at the end of the HISTORY list, a messge will be issued and you will reenter IED with the string you had before.

^X? -Is reserved. Just now it is a non-error way to leave the editor and return to it again. I think it should produce a pop-up menu of ^X functions as part of doing so, but havn't done this yet. Since the problem is that it is not IED who knows these codes, it is necessarily the caller of IED who must be prepared to display this information. Comments? This mechanism is how ^L and **Refresh** work.

There are many more control character functions that are intrinsic to IED. See the IED paper for more details.

The ^Xn and ^Xp functions, above, are what I call the "step" functions. They are the only two "motion thru history" functions available from keyboard mode via control characters. On the pop-up menu, however, you will see two "search thru history" functions that suggest the following IED keyboard mode functions:

^S -"Forward SEARCH" Lets you specify that you want to leave IED only to reenter the editor with a new command from the HISTORY list. The command is found by searching towards more recent history. You are prompted for the search argument; if you enter <cr> this means "search again".

^R -"Backward SEARCH" Is like the above only the search goes backwards thru HISTORY. The defaulting of the search argument is shared between the two searches.

The above two functions are **not** available at release 1. I earmarked these control characters to trigger the two SEARCHes, but it seems that even in cbreak mode the terminal driver eats up ^S. Since I couldn't think of more intuitive command names, and since the functions are available via the pop-up menu, I decided to drop the issue for now.

Note that when you type ^X the editor does not produce any sort of prompt. It is indeed waiting for you to type the character after the ^X, but you must know this. The best hint is that this is the state you are in when you don't see a cursor in IED and nothing seems to be happening.

5.5 BSH Built-In Functions

Aside from passing on SHELL commands for execution, the other thing that BSH does is provide for the "built-in commands" which are described in the following sections. These are very distinctly **not** your normal SHELL commands; they all have to do with manipulating the execution environment or interfacing with BSH directly.

5.5.1 The Font Change Functions

Everything that has to do with changing fonts in BSH always referres to the part of the screen called the **execution window**, described above. Other than for what happens in this window, BSH has three built-in fonts that are used for the menus, user messages, and other displays. The fourth "default fault" is loaded into the command execution window, and this is the one used by the Smx kernel to display all output generated by whatever commands you execute.

For doing font manipulations with respect to the command execution window, the following commands exist on the various menus:

SET FONT -This command lets you change the character set that the Smx kernel is using to display all I/O to the console during BSH operation. (The menu and error messages in BSH are always output with a built-in font, as is the case for IED, too). The **Set Font** function is located on the BSH pop-up menu.

If the font can be located and loaded, the kernel will start using this font instead of what it was using previously. A screen refresh is never done using the new font because no one knows what character strings were produced to paint that screen. If the font cannot be loaded, an error message is produced and the font that was in use before is not changed.

Since it is the Smx kernel that manipulates these fonts, you do indeed have multiple-font availability simultaneously; see the SF command for more details about this interface.

In P, the SET FONT command lets you do the equivalent of using a **-f** command line option in the first place. Should BSH have such an option? Also, SET FONT is also closely related with the ZOOM function, which provides a more abstract or "structured" type of font switching.

ZOOM -Lets you go from one font to another, providing for the high-level notion of "zoom" (moving **in**) and "wide angle" (standing **back**). ZOOM works with a predefined list of fonts; each time you invoke ZOOM the next font from the list is selected for an implicit SET FONT command. Initially "the next font" means

the first one in the list, and when you reach the last one a subsequent ZOOM returns you to the first one again. In verbose mode a pop-up window message is issued when ZOOM "circles around".

For ZOOM to even appear on the function menu, you must have previously defined the shell variable **Zoom_BSH_List** to contain a sequence of font file names, as in:

```
$ Zoom_BSH_List='font1 font2.ft hh1 /usr/lib/tiny'
```

The idea is that a simple ZOOM function can suffice to let the user "go along" the sequence of predefined fonts. This is especially useful when you want to provide something specific vis à vis fonts, and you want to hide from the user the names, etc, of the fonts.

For details on the mapping between font file "names" and their actual pathnames, "Font File Search Rules", below.

SHOW FONT -Causes a pop-up window message to appear which reports the full name of the currently loaded font. Just now there is no way to have BSH report the elements of the ZOOM list. Should there be? If you really want to know, use the escape-to-SHELL function and look at the Zoom_BSH_List variable described above. The **Show Font** function is located on the BSH pop-up menu.

DEFAULT FONT -When you use BSH without specifying anything about fonts, a built-in or "default font" is used. This font is actually part of the program since it has to be there to write menus and error messages, etc. Thus, as far as the user is concerned, this font does not have a name and therefore cannot be selected with the SET FONT command. The **Default Font** function is located on the BSH pop-up menu.

To allow users to return to using the default font, the pop-up function DEFAULT FONT does exactly that. In VERBOSE mode you get a confirmational message. No matter what the mode, if you are already using the default font a message to this effect is produced and no font switching occurs.

CLEAR FONTs -Is a stop-gap mechanism to let you unload all fonts from the Smx kernel. Once we have the ability to do so, we will let you selectively unload fonts. Just now **Clear Fonts** leaves you in the same state that you were, vis a vis fonts loaded into the kernel, when the machine was booted.

The above commands are modeled after those in P. By design, they are **not** as general as those provided by the SF command; we want to see at what level in the human interface we should make which functions available. The others are better left to more deeply-nested levels of detail. See the BITMAP document for more details about this interface.

5.5.2 **Some Miscellaneous** Commands

All of the other functions available in BSH are described in this section. Most of these commands are provided on the Pop-Up menu that you invoke by pointing the mouse at the execution window and pressing the left-hand (**Execute**) button.

Refresh -Does a clear and then a redraw of the screen. Just now, we redraw the entire screen so that you have a way to clear away the part that normally "belongs" to the execution of commands. Should we stick to our own part of the screen?

HISTORY -Is the way that you can recall past commands. It is fully described in its own section, below. The variable that you can define to tell BSH the size of the buffer you want it to use is History_BSH_Size. It is the value of this variable that is in effect when you invoke BSH that is used. If you want to change this during the execution of BSH, change the variable and use the pop-up function **Init History** to "start again". This causes you to loose previous history -- a feature.

INIT HISTORY -Lets you throw away all the HISTORY you have accumulated and start again. Note that there is only a limited amount of HISTORY kept in any case, but depending on the value you give in the variable History_BSH_Size, this might be an arbitrarily large number of commands. See the section on HISTORY, below, for more details.

Sh Command -Lets you enter any UNIX command without leaving mouse mode. This is distinctly different from the escape-to-shell commands in other bitmap tools because such commands are **never** executed via child processes in BSH. The **SH Command** function is located on the BSH pop-up menu and on the main on-screen menu.

VERBOSE -Lets you change the verbose/terse mode back and forth, just as in the font editor, FE, and other tools.

TTY Mode -This function is the reverse of the "$ menu" command in BSH. That is, it returns you to the mode where the mouse is inactive and commands that you type get executed by the SHELL each time you type carriage return.

BSH Tutorial -This function provides on-line documentation for BSH. In effect, this function simply calls up P with this document as the input text. When you leave P you return to where you were in BSH. There is an analogous function in IED's pop-up menu.

BLACK -Is the default and specified that you want a black background with white character displays. This refers only to what happens in the command execution window, and only to output coming from the Smx kernel.

WHITE -Is the inverse of BLACK. See above.

Off; Exit -Is a stop-gap function provided for convenience. The OFF command is executed (literally, the IOCTL that OFF uses) and BSH exits just as for the QUIT command. This function should probably be renamed "Off; Quit". Once tools like EMIN can run correctly in the execution window we will no longer have need of this function.

QUIT -Is the standard way to exit BSH. Your terminal characteristics are re-established, and a success exit to UNIX is done. In TTY mode, you can also exit the BSH with ^D.

Quit BSH -Is explicitly on the pop-up menu for convenience. It is exactly the same as the on-screen QUIT function described above.

Both forms of QUIT, above, cause BSH to output a message window which announces that BSH is terminating. This is purposefully left on your screen so that there is no mistaking when BSH is alive and when it isn't.

5.6 The HISTORY Sub-System

When you invoke the main-menu function called HISTORY, a pop-up menu appears in the upper right-hand side of the screen along with a reverse-video mouse icon which appears at the top and to the left of the menu. At this point you are invited to select from the history menu one of the commands that you have already executed, and the shell takes the action described below. The menu is arranged so that the most-recently-executed command appears at the **bottom** of the menu (so that it is closest to where the mouse is likely to be). If you invoke HISTORY and then press a button **outside** of the history menu, BSH takes this to mean that you don't want a history function and you are returned to the same state you were in before you invoked HISTORY.

Whether or not the history menu is on the screen, look first at the HISTORY menu item in the FUNCTIONs menu. If this item is highlighted, you can select something from the history menu. Otherwise, if you notice the cursor that appears as you move thru the execution window, you will see that the space taken by the history menu is now identified with the execution window. In effect, the history menu is left behind on the screen even though you cannot select from it once you have terminated the HISTORY function. The reason we leave it behind is twofold. (1) we don't view it as worthwhile to save away this part of the screen before we put up the history menu, therefore we don't have the ability to restore the screen as it was. (2) since it may be useful to see what is left of the history menu after you finish with it, we purposefully leave it on the screen (even though it gets scrolled away) so that it can be seen.

The purpose of displaying gone-by commands in the history menu is so that you can select a command string to have it:

(a) -**Re-executed** by the shell, exactly as if you had retyped the command in TTY mode.

(b) -**Picked-Up** for the purpose of being deposited into your STRINGs menu.

(c) -**Edited** before you send it off to the SHELL for re-execution, to be deposited into the STRINGs menu, and/or etc.

To remind you that these functions apply equally well to HISTORY, the mouse icon is displayed along side the history menu whenever it is valid to select something from it. As soon as you tell HISTORY what you want, the sub-system terminates, and this icon goes away. The other mouse icon on the screen does not change since the previous meaning continues to apply after HISTORY. We use two icons here to draw attention to the fact that HISTORY uses the same buttons with "the same" meaning as before (normal "mouse mode"), but that the implicit operand of these buttons is and must be something from the history menu.

Note that the so-called "history menu" is unlike a "standard menu" in that BSH takes notice of what button you press. In this way, (part of) the history menu is exactly like the STRINGs menu except that you cannot deposit into HISTORY. The only way that something gets into history is that it is a string that you passed on to the SHELL for execution. For the parts of the history menu where real functions are displayed (see the two sections below), you can press **any** mouse button to indicate that you want this function. These parts are more analogous to ordinary menus.

When you have only executed up to enough commands to fill one screen's worth of the history menu commands, the menu contains only these commands and there is nothing you can do but abort history or select one of the old commands for one of the above-described history functions.

After you have accumulated enough history to require more than one menu-full to display it, the history sub-system becomes much more elaborate because now you can "walk thru" your old history, a menu-full at a time. Once you are positioned in mid-history, you have the possibility of walking both forward or backwards. If you have the possibility of walking, then you also have the possibility of SEARCHing. All of this is detailed in the following sections.

5.6.1 Traversing Your HISTORY

At the top or bottom of a history menu, you may see a menu item box that has pointing arrows and the text "for OLDER history" or "for more RECENT history". If you select such a box with **any** mouse button, you will see another menu full of history. The menu will contain either more-recent or older history, depending on which one you select.

When you cannot go any further thru HISTORY than where you are, the menu item that would offer you the choice simply doesn't appear. There is no way, just now, to jump arbitrarily to the beginning or to the end of the history sequence of menus. If you exit HISTORY and rebegin, you will always be at the

most recent menu-full; to actually get to the oldest HISTORY menu you have to search or step backwards.

Note that the beginning of the HISTORY doesn't necessarily correspond with the first command you executed. The HISTORY buffer is a fixed size no matter how that size is established, and the commands that are the oldest are discarded when the buffer is full and more recent commands are to be memorized. See the HISTORY function, above, in the section called "Some Miscellaneous Commands" for the details on how to "grow" and "reinitialize" your HISTORY buffer.

5.6.2 SEARCHing Thru HISTORY

Another way to "traverse" your history, for the purpose of repositioning the selection menu, is to invoke one of the boxes that say "history SEARCH". They appear with upwards-pointing or downward-pointing arrows, at the top or bottom of the menu, depending on whether the search is to be backwards - thru older commands, or forwards - towards more-recently-executed history.

You are always prompted for a search argument. If you enter <cr>, and if you have already searched once during this session, then the same operand you used last time is defaulted.

A history SEARCH **never** considers the commands in the menu you are looking at. A backwards SEARCH starts with the command line immediately above the top one that you could see before; a forward SEARCH starts with the oldest command that is more recent than the bottom one on the visible menu when you issue the SEARCH.

When no command can be found to contain the given substring, a message is issued and you remain in the HISTORY sub-system looking at the same place in the menu as before. If a command is found to contain the string, a new history menu is drawn with the found string in the **bottom** position of the menu.

5.7 How It Works

Since BSH is really an experiment in human interfaces, especially with respect to personalizing menus, etc, some words on how it all works seem appropriate...

5.7.1 The Shell Interface

It was possible to build a "shell interface" only because of a few crucial goodies in the Smx kernel. The main one is the ability to execute **any** UNIX process in such a way that when it does I/O "normally", the output appears in a readable font on the numelec screen, and input can be captured by a program. As

associated goodie is the ability to have what I call a "scrolling region"; this permits BSH to have your SH commands running in the main part of the screen, whilst keeping menus and other information (date and time, etc) on its own part of the screen.

The IOCTLs described in the BITMAP document supply what is needed here. The bitmap SHELL simply sets a scrolling region appropriately, and loads either a default or a specified font into the SH "execution window" by loading it into the kernel.

To avoid a maintenance nightmare, both BSH and SH are generated from the same set of sources. If you compile these sources with no special options, the result is SH - literally the SH that is distributed with Smx. If you compile these same plus a dozen more source files with the CC option:

-DBOTH_BSH_AND_SH

the resulting program is BSH. It is these dozen or so files that we will distribute to users who want something to start from to build their own interface to an arbitrary bitmap application. See "Further Information", below, for the details.

In a similarily rigorous way, the interface between the real SHELL and BSH is extremely well defined, too. BSH manipulates numerous data structures (the STRINGs menu, the HISTORY buffer) and interacts with the user via the keyboard, IED, the mouse, and/or some menu, but eventually comes up with a character string to be executed. Using conditional compilation in the original SH sources, I simply supplanted the lowest-level get-command-line primitives so that they now "real" their input from the buffers that are filled by the interface.

The only other change to SH was the addition of the menu command, which was added in a manner completely parallel to READONLY, which is based on EXPORT and shares a lot of code with it. I also added some predefined variables, like PATH, to accomodate font file search rules, HISTORY buffer size determination, and the ZOOM list variable.

5.7.2 IED String Editor

The so-called "Icon chain editor", IED, is a major part of BSH. Without it I'm not at all sure that writing the shell interface would have been worth it. I thought of building-in IED, but VME experience has shown that such a tool can be very useful if it's generally available.

A unique aspect of BSH and IED co-habitating is that they have mutually exculsive pop-up menus. This is sharply contrasted with the two-mode model in FE where parts of a single menu are shared and parts are mutually exclusive. It is not clear which model is most effective. Neither presents a significant programming difficulty; the issue is more one of what is the least confusing for the user and for the documentor.

See the IED document for more technical details on how it works.

5.7.3 Font File Search Rules

In order to present this issue of "font file search rules" consistently, I have written about it in the document (called BITMAP) that describes the whole tool set available from INRIA that work with the Numelec bitmap. For BSH's usage of fonts, suffice it to say that BSH follows the rules described in this other document. Note that this type of "load font" is explicitly not the same as what P and other bitmap tools do. Here it is the Smx kernel which loads the fonts, and it is then used in the normal terminal output associated with the console. This allows BSH to run arbitrary UNIX processes in this "scrolling window" without them having to know how to do "bitmap I/O".

To find out the name of the font that is being used by BSH, see the SHOW FONT command. To return to using the built-in font that BSH loads into the "SH execution window", see the DEFAULT FONT pop-up command. The three fonts used in BSH for messages, menus, history, and IED-editing are modified versions of fonts taken from the standard (distributed) font set.

5.8 Further Information

For more information on using BSH, please refer to the paper that describes the whole set of tools for the bitmap. Since BSH was written explicitly to demonstrate using the Numelec software, the mouse, and particularily the menu package, we will provide source listings of the interface part of BSH to anyone wanting to develop bitmap software for Smx and the Numelec.

For usage information on BSH explicitly, you can get a printed version of this text by simply copying the ASCII file:

$BITMAP/numelec/Numelec_Doc/bsh.doc

if it was put there on your system. There is a link to this file in the usual "../Numelec_Doc" directory where "all" the bitmap documents are stored. (See the top-level .profile for the definition of DOCs). To be able to use the TUTORIAL function in BSH, you must either literally have this document stored in the above file, or you can define and export the shell variable before you invoke BSH,

```
$ Numelec_Doc=/mnt
$ export Numelec_Doc
```

then the string "/mnt" is prepended to the "bsh.doc" pathname above by BSH. Briefly, you can install the bitmap distribution disk any way you want to, but you cannot alter the relative position of the documentation directories. Like the symbol BITMAP, the corresponding documentation symbol, Numelec_Doc, points to the top of the tree so that it can be used by several tools independently. Note that the on-line version is not paginated; this makes it much better for looking at on-line with P. If you want a copy of the paginated,

type-set version of this document, ask the person who installed BSH on your system, contact the author, or Michel Loyer at INRIA.

Any suggestions on how to improve BSH or this text would be appreciated.

Contact: Kevin Pammett, INRIA, Rocquencourt. (3) 954-9020 poste 3160.

[end of BSH tutorial text – updated as of 27 March, 1984]

CHAPTER 6

A Callable String Editor

In Retrospect: This is my second prototype for a "callable string editor". The first was in VME at DEC, where the editor was limited to a particular part of what you see here (because of the VAX/VMS software architecture). I believe strongly in this aspect of human interfaces: the **same** editing mechanism that pops up in a variety of circumstances to let you change icon strings irrespective of what the string is to be used for.

However, it will require time and dedication to the spirit of IED for it to become wide-spread. IED appearing as part of BSH is only the beginning. The whole point of the pop-up string editor is that it be integrated into most of the highly-interactive tools that we use. Such is certainly not the case as of this writing. However: Petit à petit, ça avance!

The rest of this chapter contains the IED document verbatim.

IED – The (Callable) String or ICON-Chain Editor

(Tutorial)

Kevin Pammett

I.N.R.I.A.
Rocquencourt, France.

6.1 SYNOPSIS

result = do_edit(at_x, at_y, string)

IED is a callable string editor. That is, it is a set of routines that you can link with your bitmap applications program when, for an arbitrary reason, you want to offer the user the possibility of editing a character string as part of using your application. The IED editor is self-contained, and it provides a highly-polished menu-driven, "bitmap" and keyboard interface designed for the purpose of editing a chain of icons, of which a sequence of characters is a typical example.

Note: IED is not a program. An example of its use within a UNIX bitmap program is BSH, the "bitmap SHELL" that is described in a companion paper. IED's editing notation has been explicitly modeled after that found in EMIN,

EMAX, and numerous screen editors of that ilk. The idea of having a built-in, callable, string editor comes from the "Vax Made Easy" toolset called VME.

This version reflects IED at "baselevel 1", which coincides with version 3.2 of Smx. This is the first public version of this document. Since a "callable string editor" is somewhat of a research problem, this is a "working design document" for IED; the text contains numerous questions intended to further the development of the interface via user feedback.

6.2 Rationale

IED is a specialized string editor which can be called up by a bitmap tool given a string to let the user edit. The editor finds a place on the screen to "pop-up", having stored away the screen first, much like a pop-up menu or window, and presents the user with a consistent interface for changing strings.

For now, IED is a "goodie" that appears in BSH. The whole point of IED, though, is that if it is used as an off-the-shelf item in most major bitmap tools, that users will then have learned how to edit ad hoc strings only once, yet they will have this flexibility in most of their interactive intensive tools. This is not to say that IED should replace screen editors like EMIN. Not at all. Text editors, font editors, and graphics editors are here to stay. IED will never replace them; instead it will simply take its place along all the other specialized editors.

6.3 Presentation

What you see when IED comes up is the string you want to edit, displayed in a rectangular box on the screen, with somewhat-strange-looking icons attached to each end of the box. These icons are what I have termed the **scroll icons**, modeled after the PERQ and STAR notions of having the ability to scroll editing windows or menus.

The intent is that these icons **not** be there under normal circumstances. As of this release, however, the notion of horizontal scrolling within the IED editing window has not been implemented. Therefore I decided to leave the icons there because if you put the mouse in either of these icons and press any button, you will be told that SCROLL is not yet implemented.

The **intent** is as follows: When you first come up in IED, it makes a window "big enough", if that is at all possible, to let you do all the editing you're going to do without having to fuss with anything. Of course IED could simply make a window of maximum size and be done with it, but that is extremely unelegant and not very-well human engineered.

Normally, then, if the string can be displayed on the screen, you don't see either of the scrolling icons; you simply see the kind of rectangular box that one might expect.

Suppose, then, that you go to the right-hand end of the string and start typing. Purposefully, there is room left for what we expect is a reasonable amount of addition to the string. As you type, your cursor advances towards the right-hand end of the window, and eventually you will be "close enough" to the end that something must be done. This is when the whole string should start moving to the **left**, hiding the left-hand end of the string as you type. At this moment the left-hand scroll icon appears, to show you that some of the string is hidden, and to let you know that if you want to scroll the window that way you use the mouse in conjunction with the icon to do it. The icon can be thought of as a simple-minded menu which appears automatically only when you need it and then disappears again when/if it becomes unnecessary.

If you add to the left-hand end of a string, a similar situation comes up when the string grows "too large" for the window. At this point the string simply goes off the right-hand end and the icon appears just as before except that you are invited to scroll the other end of the window, now.

My theory for all this is as follows: (1) none of this will happen very often because a good-enough heuristic is used to guess at the size of the window in the first place. (2) even when part of the string gets hidden, the user is assured that nothing has gone wrong because the icon nicely appears. (3) Since IED is specialized, we can suppose things about what the user is doing. In the case of BSH the user probably wants to execute the string as a command. In this case he will probably never even use the scrolling function because he will know perfectly well what the string contains; he only wants to add on to the end of the command before he executes it. Hiding something that is unlikely to be used is much simpler to the human than going to great lengths, drawing attention to the "wrong" thing, all for nothing.

Well, all that is very nice but none of it is proven. Perhaps the magic of this from of presentation will be implemented in the next version and we will see how "comfortable" it is.

Of course the biggest question, alternative to this, remains. If we don't do something like the above, then how do we have a specialized one-line editor on a screen that has a fixed width? One answer is switching fonts when things get desperate. This only delays the question, though.

Comments?

For the practical meanwhile, see the ^L command for a way to "grow" the window gracefully. This is the same as the **Refresh** function documented below. For a more in-depth explanation of entering and leaving IED, see the section "Leaving The Editor", near the end of this document.

6.3.1 Keyboard Versus Mouse

Except for the limited case of deleting text, changing a character string necessarily involves either moving around existing text or entering new text. We have provided for two of these three classes of operations; the one that is missing (just now) is that of moving around text. Provision has been made for it via earmarking a ^Y command and some marking operations, but none of this is really implemented in the first version.

Deleting text is easily done with the MODIFY function, described below. For the most general approach to changing your character string, though, see "Keyboard Mode", below, where we describe all the "control characters" that let you perform EMACS-like operations. The reason that IED is not "just a one-line EMIN", though, is that heavy use is made of cursors, pop-up menus, and special-purpose modes or functions.

6.3.2 Cursors Usage

When IED is invoked, the user is first presented with "mouse mode", in which a special "floating" cursor is used to locate exactly where in the character string a change is desired. A similar cursor is used in MODIFY mode where you can delete characters, replacing them with either spaces or deleting them entirely, all without using the keyboard whatsoever. In general, the cursor follows the mouse and its form changes to let you know what you can do. When you are in keyboard mode, there are two more cursors that are used to distinguish **change** versus **insert** mode. Outside of the IED editing box, two different cursors are used, depending on whether you are in "no man's land" or whether you have invoked a pop-up menu.

6.3.3 The Function Menu

Unlike most other bitmap tools, IED does **not** put up an on-screen menu. This is becaue (a) one is not necessarily required, and (b) there may already be such a menu on the screen because IED is usually called up in a nested fashion by another tool. Thus, when IED comes up, the only thing it does is to make its own editing window by saving away the contents of the screen so that it can be restored later. If you see an on-screen menu during IED operation, you cannot select functions from it and you should consider it as simply not being there. IED makes no attempt to find out if there is such a menu, to change it, to let you know that this area of the screen is different from any other, etc.

6.3.4 The Mouse Icon

To the right of the function menu you will see an **icon** that is meant to tell you the meaning the mouse buttons. In general, this is the same icon that is used in FE and in other bitmap tools except unlike these other tools, IED (and BSH) have a mode in which the mouse is **inactive**. For these periods, the "mouse icon" is replaced by something that resembles it but is meant to tell you that pressing the mouse or moving it will not have any effect. While we're at it, we present a message about what you have to do to get back into "mouse mode", and for what reason you are NOT in mouse mode already. As for the other bitmap tools, during the operation of IED the phrases attached to the buttons are likely to change radically.

If you make some sort of error while using IED, a message is usually produced in a pop-up window. In general, this window is reverse videoed so that you will take notice of it, and a time delay is built-in to IED so that the window will automatically go away without the user having to do anything. See the

document on P for further comments on this issue.

6.4 Mouse Button Functions

There are two different states that you can be in with respect to the mouse buttons in IED. The first one is active when you invoke IED; the other applies only to the nested function called MODIFY.

During the normal operation of IED, pressing **any** mouse button outside of the IED editing window will cause a pop-up menu to be displayed. (See below). **Within** the editing window, the following mouse button functions are offered:

INSERT -Puts you into keyboard mode and you see an **insert** cursor, whose form is meant to show you the meaning of insert. Each time you type a non-command character, it is inserted into your IED buffer and all characters to the right of it are moved to the right to make space. You leave this mode with <cr>, ^Z, or one of the ^X functions. See below.

CHANGE -Puts you into keyboard mode and you see a **change** cursor, whose form is meant to show you the meaning of change mode (otherwise known as **overstrike**). Each time you type a non-command character, the character that the cursor was on is deleted, and the one you typed is inserted into your IED buffer. If the new and replaced characters were the same width, none of the other characters move. Otherwsie you will see some bit-position movement, but the number of characters in the string does not change. This mode is especially useful for changing arguments to commands, parts of file names, and etc. You leave this mode with <cr>, ^Z, or one of the ^X functions. See below.

DONE EDIT -Is the expected or "success" way to leave IED. What this means is determined by the caller of IED. For the BSH case, **Done Edit** means that you continue on to do what ever you started; be it to execute a command, to deposit it into the menu, etc.

If you point outside of the IED editing window and press any button, you will get the pop-up menu described in "The Pop-Up Menu", below. One of these functions is MODIFY, which is designed to let you edit the string without using the keyboard. When you invoke MODIFY, yet another cursor is produced - to remind you that you have changed modes and to signal you to note the mouse icon which also changes.

In "MODIFY mode", the following mouse button functions are offered:

DELETE -Is the traditional form of "delete a character". You point to the character that you no longer want, and press the DELETE function. The character will disappear and all characters to the right of it will move to the left one position.

White-Out -Is a new form of delete, invented especially for the specialized IED editor. You point to the character that you no longer want, and press the **White-Out** function. The character will disappear and the space it was taking up is replaced by white space (pixel blanks). Internally, the deleted character is replaced by a real blank, but rather than use the actual blank character for display, a variable-sized one is simulated so that no other characters in the string move in any way. This is the way to take something away from your command without incurring the side-effect of having the space disappear.

Done Modify -Is the only way to return to normal IED editing mode. Since this is the same "done" button that you need to leave IED, you will see that clicking this button **twice** is a fast way to be in MODIFY mode and to get out completely.

All of the above MODIFY functions must be triggered by pressing the indicated mouse buttons while you are in the region of the IED editing window. If you point outside of the window and press either of the editing buttons, an error message will be issued. On the other hand, if you invoke **Done Modify** outside of this window, you will nevertheless return to normal IED editing mode. In no case can you be in MODIFY mode and get the IED pop-up menu simultaneously.

6.5 Keyboard Mode

Other than the limited case of MODIFYing your string, the central model in IED is that you will (a) point to the part of the string you want to change by moving the mouse, and (b) enter **keyboard mode** and then use control characters to specify the editing functions you want.

The next three sections describe these control character editing functions in detail. When you enter keyboard mode the mouse icon will change to tell you that the mouse is no longer active. When you enter <cr> in this mode you will return to mouse mode and can then get the pop-up functions, leave the editor, or reposition the cursor and reenter keyboard mode.

6.5.1 The Cursor Motion Functions

Once you go into keyboard mode in IED, moving the mouse is no longer a way to point to a character. To help show this to the user, IED uses two keyboard mode cursors, depending on whether you are **insert** or **change** mode. The former is a wide up-pointing arrow, while the latter is a large **X** which is purposefully placed over top of the character.

The following commands have to do with **cursor motion** in keyboard mode.

^A -Control-A moves your **insert** or **overstrike** cursor to the beginning of the character string. (Actually, to the left-hand end of the editing window. I havn't determined yet if this implies a scroll of the actual characters being edited. Just now, IED doesn't scroll at all.

^E -End. Control-E, moves your cursor to the end of the editing window. Like Control-A, I havn't determined yet if this implies a scroll of the actual characters being edited.

^F -Forward. Control-F moves your cursor to the right one character. In insert mode the cursor and the character swap positions so that we don't have the "flicker" of moving the right-hand part of the string left and then right again.

^B -Backwards. Control-B moves your cursor to the left, back one character. In insert mode the cursor and the character to the left simply swap positions.

The above commands are modeled after those in EMIN. We don't have the "word motion" equivalents. Should we? My feeling is that we don't need this because most "pointing" in IED is done with the mouse. This is IED's advantage over "keyboard editors"; what could be more natural than to point to characters that you want to change using the mouse?

6.5.2 Deleting Characters

One way to delete characters is to use the MODIFY pop-up function described above. If you are in keyboard mode, however, it is much more convenient to use one of the following control characters that all have to do with deleting characters or strings.

^D -Control-D deletes the character to the right of an insert cursor, or the character that the cursor is on, when you are in change or overstrike mode.

^H -Control-H (or Backspace) Is the traditional way to delete the character to the left of the cursor. The UNIX notion of ' ' is not used because almost no one choses that over BACKSPACE when they have a terminal on which that makes sense.

^K -Control-K. Deletes all characters in the editing string to the right of the cursor, including the character the cursor is pointing to.

^U -Control-U. Deletes all characters in the editing string to the left of the cursor, not including the character the cursor is pointing to.

The above commands are modeled after those in EMIN, EMACS, UPE, VI, and other editors of the same geneology. Like EMIN, the IED editor does **not** take into account your current STTY defaults.

6.5.3 The **Mode Change** Functions

Other than using the mouse to trigger pop-up functions, the following **keyboard mode** control characters also have to do with mode changes. Some of the functions described in this section can also be triggered by pop-up menu functions.

^C -This command lets you change from **insert** to **change** mode or vice versa. It is really a life saver when you entered keyboard mode in one mode, but then decide you wanted the other mode.

^Z -This is another way to leave IED successfully. It is exactly equivalent to the **Done Edit** function, except that you invoke this function from keyboard mode while the former is only available from "mouse mode".

<cr> -Carriage-RETURN is the way to return to "mouse mode" when you were in keyboard mode. If what you wanted was to leave IED completely, see the ^Z command, above.

^L -Redraw. Actually, this function causes you to leave the IED editor and re-invoke it again with the current (changed) string. The effect is that the size of the editing window is recalculated so you have a new maximum size string to edit. Just now, this is the only way to "grow" or "shrink" your editing window.

Whenever you leave keyboard mode via carriage return, you return to IED mouse mode. Use ^Z if you want to return directly to the caller of IED.

6.5.4 Leaving The Editor

The standard way to leave the editor is to use the **Done Edit** mouse button. When you do this, you are effectively saying to the caller "do whatever is the default action" with the changed string. That is, this is the "success" way to leave the IED editor. For the "not success" way to leave IED, see the **Quit IED** pop-up function described below.

To make things slightly easier, there is also a way to leave from keyboard mode: you type ^X followed by one of the characters described below, or you type ^Z, the keyboard way to leave IED via the "success exit". See the **Done Edit** function for more details.

^Xa -This is the `a' exit; it is the caller of IED who interprets the `a', which can be **any** character whatsoever. Just now, if IED is called up by BSH, you will get a message about the "return code" `a' being unknown and you will find yourself back in the editor again. It is BSH that determines this, not IED.

^Xm -Is an application-independent way to leave IED and to return something significant to the caller, without IED having to know about specific types of "return values". The character `m' typed immediately after the ^X is passed back to the caller, no matter what. In the case of IED being called up by BSH, the return code `m' stands for "menu" and you will have invoked the BSH function called "deposit into menu and execute". See the BSH document for more details.

^Xp -Previous. You leave IED and enter again, this time editing the previous (History) command. For applications where there is no notion of a sequence of things to edit, the caller of IED will simply report that this "exit sequence" does not apply and you will reenter IED again so as to not loose what you were editing. The same applies for ^Xn.

^Xn -Next (forward). You leave IED and enter again, this time editing the next [more recent] (History) command. See above.

^X? -Is reserved. Just now it is a non-error way to leave the editor and return to it again. I think it should produce a pop-up menu of ^X functions as part of doing so, but havn't done this yet. Since the problem is that it is not IED who knows these codes, it is necessarily the caller of IED who must be prepared to display this information. Comments?

The above commands are actually interpretted by the caller. Some of them are meant to be "standard", like the next and previous notions which usually will apply to whatever you can edit with IED. Other ^X exits are completely application-dependent.

Note that when you type ^X the editor does not produce any sort of prompt. It is indeed waiting for you to type the character after the ^X, but you must know this. The best hint is that this is the state you are in when you don't see a cursor in IED and nothing seems to be happening.

If you type a character following the ^X that is **not** expected by the caller, a message is produced and you are returned to IED with the same string you had when you left. This is a protection mechanism to help users who didn't mean to type ^X. Note that the ? character is the only one that is treated specially here. If you leave with ^X? you get the same treatment as if the character hadn't been known, but you don't get a message about it not being know. This is how ^L or **Refresh** work.

6.6 The Pop-Up Menu

The pop-up menu in IED has a fairly important role to play since there is no other menu to work from. You invoke the pop-up menu by being in normal IED mouse mode (i.e. not MODIFY), by pointing outside of the IED editing window, and by pressing **any** of the mouse buttons.

Some functions appear only on the pop-up menu, but most are at least partly described elsewhere because quite often there is a "keyboard" way to call the function as well as a "mouse" way to do so.

In any case, the following entries appear on the pop-up menu.

Cancel Pop-Up -If you actually select CANCEL POP-UP, the pop-up menu disappears and you return to IED just as if you had not invoked a pop-up function. There is no difference between selecting this function and moving the mouse outside of the pop-up menu and then pressing **any** mouse button.

Execute & Deposit -No matter how you entered IED from BSH, this function is how you do both an "execute" of the command string, and a "deposit" of it into the menu. It is equivalent to the keyboard mode command: ^Xm If you did not get the string in the first place from the menu, you will be prompted to where to deposit it.

Execute String -No matter how you entered IED from BSH, this function lets you send the edited string off to UNIX for execution. Is equivalent to the keyboard mode command: ^Xe

MODIFY String -This function lets you modify the IED string without using the keyboard. The functions offered are very limited because there is not a lot you can do without a keyboard. Via the mouse buttons you are able to **white-out** or **delete** characters. See the section "Mouse Buttons", above, for more details.

Refresh -This function actually causes you to leave IED, which implies putting back the screen, after which IED is reinvoked with the changed string. The side effect of this, besides a redraw, is that the maximum size of the IED editing window is recalculated. Thus, if you want to make your window larger or smaller, the **Refresh** function is sufficient.

Deposit into Menu -This function is useful when you entered IED by some BSH function other than "Edit Menu" and you later decide that you want to have the edited string deposited into the menu. Unless you did enter by "Edit Menu", you will be queried for where in the menu you want the string to be deposited. This function is equivalent to the keyboard mode command: ^Xd

IED Tutorial -This function provides on-line documentation for IED. In effect, this function simply calls up P with this document as the input text. When you leave P you return to where you were in IED.

DONE IED -Is the standard way to exit IED. A success exit is done to the calling procedure, and it takes whatever action it normally does with the edited string produced by IED.

QUIT IED -Is the **non**-standard way to exit IED. Things should be exactly as if you hadn't called the editor in the first place.

Most of the functions on the IED pop-up menu really have to do with the specific BSH reason for calling up the editor. For this reason, it seems that in future uses of IED the pop-up menu mechanism will be supplied and documented by that caller, not by the editor. In any case, I have put all this here to help stress the autonomy of IED with respect to BSH.

No matter how you get into IED while using BSH, you always end up editing a SHELL command. Whether or not this command has ever been executed, it is still considered part of BSH's notion of HISTORY. Therefore, there is always the notion of **time** with respect to any SHELL command. This is to say, except for the boarder line cases, there is always a command before (time-wise) and after (more recent) than the one you editing.

When circumstances are right, then, the following functions may appear on the IED pop-up menu. To minimize the complexity of the user interface, the functions do not appear on the menu if invoking one of them would simply cause the editor to produce an error message.

Forward SEARCH -Lets you specify that you want to leave IED only to reenter the editor with a new command from the HISTORY list. The command is found by searching towards more recent history. You are prompted for the search argument; if you enter <cr> this means "search again".

Backward SEARCH -Is like the above only the search goes backwards thru HISTORY. The defaulting of the search argument is shared between the two searches.

Step Forwards -Lets you specify that you want to leave IED only to reenter the editor with the next more-recent SHELL command. This is to say, **Step Forwards** lets you go ahead thru HISTORY. This function is exactly equivalent to the keyboard mode command: ^Xn

Step Backwards -Is analogous to the above except that you get to edit the next **older** SHELL command. This function is exactly equivalent to the keyboard mode command: ^Xp

Of the above, only the "step" functions are available from keyboard mode via control characters. That is to say, the "search thru history" functions are **only** available via the pop-up menu. We could have invented control characters to trigger the two SEARCHes; is it worth it? I thought of using ^S (search forward thru history) and ^R (backwards) here, but it seems that even in cbreak mode

the terminal driver eats up ^S. Since I couldn't think of more intuitive command names, and since the functions are available via the pop-up menu, I decided to drop the issue for now.

6.7 How It Works

Since IED is an experiment in tightly coupled but independent software tools, some words on how it all works seem appropriate...

6.7.1 Font Usage in IED

When you invoke IED you see a large editing window with your string to be edited displayed in large characters. By design, IED uses a particularily large font to display this string, and there is no way just now for the caller to change this. (In actuality, it is at LINK-time that this binding is determined. IED only knows that the LINK-time name of the font is IED_FONT; theoretically one could replace IED's font whenever the editor is relinked with a (different?) application).

The IED_Font used is variable-sized and contains special characters and cursors. All of these characters fit within a maximum height, but the width of any character is arbitrary. In essence, IED is really a high-level manipulator of a linked list of icons.

When the IED pop-up menu is displayed, or when an error message is issued, a different (smaller, more regular) font is used. This is the one supplied by the BITMAP menu package.

Neither of these fonts are loaded into the kernel; instead they are compiled and linked into IED directly. This is like FE and P but unlike the part of BSH that has to do with the execution window.

6.7.2 IED String Editor

The idea of a "callable" string editor comes from VME, the Vax-Made-Easy toolkit that I prototyped for VAX/VMS. Here there was a one-line string editor that could be invoked to edit hidden buffers of strings that could eventually get passed on to the command interpreter for execution.

For such an editor to be workable, it must be invokable efficiently from a program, and it must be able to transmit its results - string results - back to the caller. In the UNIX world this translates to saying that you must link IED into your program. Well, that's simple; it presents only organizational problems.

The second constraint is that IED must be transparent. Here the save and restore screen Raster-OPs supplied the needed magic.

It is mostly the Raster-Op library that makes IED possible. Indeed, it was an exercise in using the Raster-OPs to build IED - practically everything that IED does is done with a Raster-OP.

There is no inherent reason why a non-callable - ie. a directly invokable - version of IED cannot be provided. Especially in conjunction with UNIX's command-line intrinsic functions - the back-quote convention - an IED program could be invoked by **any** shell as in:

```
x=`ied $x`
```

where the IED program simply prints the resulting string on its standard output after the user does DONE EDIT. This form is not provided just now cause I didn't have time, and because I think it more important to promote the callable form of IED. Comments?

6.8 Bugs and Limitations

There are two glaring problems with the current implementation of IED.

First of all, IED tries its best, but nevertheless completely fails to put back the screen properly when you leave the editor. Here I use the same combination of Raster-OPs that all the other pop-up menus and windows use, but from some reason it doesn't work here. One difference is that this is the only case where I try to get the save memory from **malloc**. I don't see how this memory could be distinguished from memory allocated statically, so suspect that the problem lies elsewhere. Sorry.

The other bug only appears when you try to edit a string that is too large. For the details on this, see the section, **Presentation**, above.

6.9 Further Information

Since IED was written explicitly to be "callable", it is important that we work out an interface that really is general enough but simple enough to be usable at the "call" level. In this sense, the "IED package" is very much like the "menu package". We will provide source listings for IED to anyone wanting to develop bitmap software for Smx and the Numelec providing that it is understood that the interface is likely to evolve.

For usage information on IED explicitly, you can get a printed version of this text by simply copying the ASCII file:

$BITMAP/numelec/Numelec_Doc/ied.doc

if it was put there on your system. There is a link to this file in the usual "../Numelec_Doc" directory where "all" the bitmap documents are stored. (See the top-level .profile for the definition of DOCs). To be able to use the TUTORIAL function in IED, you must either literally have this document stored in the above file, or you can define and export the shell variable before you invoke BSH, as in:

```
$ Numelec_Doc=/mnt
$ export Numelec_Doc
```

then the string "/mnt" is prepended to the "ied.doc" pathname above by IED. Briefly, you can install the bitmap distribution disk any way you want to, but you cannot alter the relative position of the documentation directories. Like the symbol BITMAP, the corresponding documentation symbol, Numelec_Doc, points to the top of the tree so that it can be used by several tools independently. Note that the on-line version is not paginated; this makes it much better for looking at on-line with IED. If you want a copy of the paginated, type-set version of the document, ask the person who installed BSH/IED on your system, contact the author, or Michel Loyer at INRIA.

Any suggestions on improving IED or this text would be appreciated.

Contact: Kevin Pammett, INRIA, Rocquencourt. (3) 954-9020 poste 3160.

[end of IED tutorial text -- updated as of 23 March, 1984]

CHAPTER 7

An Unsuccessful Port of MIT's Window System

In Retrospect: Sometimes one's failures are more enlightening than one's successes, and this chapter is very definitely a case in point. I spent 2-3 months trying to port MIT's window system to our SM90 UNIX kernel, and never did get out enough of the bugs to produce someting usable. I learned a lot in trying, though.

One thing is for sure: window systems are here to stay, and there's lots of research yet to be done to upgrade our notion of human interfaces to take into account the parallelism and other issues that are unique to working on several things at once. See the "Bitmap Shell (BSH)" chapter herein for some relevant research about "research directions to persue **after** you've got your window system working".

The rest of this chapter contains the WSYS document verbatim.

The SM90 Window System For UNIX

(draft version)

Kevin Pammett

I.N.R.I.A.
Rocquencourt, France.

7.1 Abstract

This paper is a rewrite of the MIT paper on their NUnix window system. Consequently, it currently describes a combination of their UNIX-based Window System, developed for use with the MIT-RTS NU Personal Computer, and ours, which is for the SM90. As the port to the SM90 procedes, I intend on upgrading the paper to accurately reflect the UNIX/SM90 implementation. Until further notice, it should be noted that the system described herein does NOT exist and therefore cannot be distributed.

On both systems, the window system is implemented via a set of device-driver routines in the UNIX Version-7 kernel. It comes complete with a user interface program which provides the notion of multiple concurrent windows, each with their own shell.

Portability across UNIX implementations, provided a high-resolution display is to be used for output, so far seems to be reasonably straightforward.

7.2 Overview

The SM90 implementation of "windows" contains two separate parts: (1) the part of the kernel which has been modified to provide the equivalent of system calls by which applications programs can manipulate windows, and (2) an applications program called the **window manager** via which users get to use and manipulate windows at a much higher level. Most of this paper documents (1), which is what I mean when I say "the SM90 window system". For now, only slight mention is made of (2) which will be further documented when time permits.

The SM90 UNIX window system has the following overall characteristics:

(1) Windows are independent UNIX teletype devices which respond to all **TIOC**xxxx standard ioctl-calls and can be frozen or thawed independently.

(2) Windows also respond to a set of **WIOC**xxxx ioctl-calls used to manipulate their "windowness" attributes such as location, size, font-usage, exposure, keyboard status, and others.

(3) Every window is associated with the raster-scan-display device upon which it is displayed. At "init" time, a parent window is associated with the Numelec screen; all subsequent windows are created within the context of this initial window.

(4) Windows may or may not have a one-to-one relationship with UNIX processes; application programs are free to impose their own structuring on windows which are otherwise viewed as "flat" by the SM90 UNIX kernel.

(5) Every window has an independent multi-font map; the actual fonts are stored in the kernel and are therefore shared by all users.

(6) Every window belongs to a process-group to which a signal is sent whenever the window changes size or location, becomes exposed or covered, gains or loses current keyboard window status, etc.

(7) Only the super-user has control over all windows in the system, otherwise a window can be manipulated only by a process in its process-group or by a process with write permission on the window.

(8) All windows, upon creation, are entered in the SM90 UNIX file system as character-special devices in the directory /dev/wdev with a filename specified by their user-supplied label.

(9) Windows are dynamic in that their location, size, exposure, font-map, etc. can be modified at any time under user-program control.

When a window is created, it can be used in a variety of ways by the process that created it. Most commonly, windows are created and processes such as the UNIX **shell** or the **Emacs** editor run in them. Under such circumstances, each window serves as a separate UNIX teletype with an independently scheduled process(es) running in it. When a window is covered, output to it can be saved in a buffer and displayed when the window becomes exposed again, in this way a process will not be halted because it wants to output to a covered window. Due to hardware and efficiency considerations on the NU, only the ascii-image of a window is saved. Programs on the NU such as EE (a screen-oriented text editor) and Draw (a circuit design editor), which manipulate their window displays in a special manner, use the window-signal mechanism to wakeup and refresh their displays when their windows become exposed.

7.3 The SM90 Machine

The SM90 window system is implemented on the SM90 computer designed by CNET, the Centre National d'Etudes et Télécommunications. Due to modifications to the UNIX kernel, the window system is available only with the version of UNIX (called "Smx") distributed by INRIA.

The SM90s currently being used at INIRA, CNET, SEMS, IMAG, and other sites typically consist of:

(1) a multi-processor architecture including at least one MC68000 CPU with custom memory-management hardware, and associated I/O processors.
(2) One-half to several megabytes of main physical memory. (Maximum is 7 meg, I think).
(3) a high-resolution (1024 x 780 point) raster-scan display made by Numelec. (Note: this is the opposite shape of the MIT screen in which they chose the upright "page" style rather than the traditional "horizontal" format).
(4) at least a 20-Megabyte Honeywell-Bull disk (10-Megabytes fixed and 10-Megabytes removable). Several SMs have much larger disks.
(5) a keyboard (also made by Numelec) which is somewhat different from the "AI" or "Micro-Switch" keyboards in use at MIT.
(6) a mouse pointing device, also provided by Numelec, with three mouse function buttons.

In general, the memory on the "Numelec card" is used exclusively for bitmap software; it is **not** included in the general physical memory that the UNIX kernel is concerned with allocating and using for user processes.

The NU machines at MIT are being utilized in several circuit design projects and in the development of a powerful multi-font text editing system and a versatile drawing facility. In addition, the NU's serve as a testbed for the development of new operating system concepts.

The SM90 is a more modern machine which has already served as a test-bed for numerous hardware and software projects, including a tightly coupled multi-

processor systems research project. Now that SM90s are generally available, it is impossible to say inclusively what they're being used for.

7.4 Window IOCTL Calls

User program creation and control of windows in the SM90 UNIX window system is accomplished via a set of ioctl-calls. These calls allow a user-process to: (1) make and initialize a new window, (2) draw or erase a selected window, (3) obtain the current state of a selected window, (4) modify the current state of a selected window, (5) select and manipulate the fonts utilized by a selected window, (6) read the state of the mouse device in a selected window, (7) obtain the current state of the raster display to which a selected window belongs, and (8) switch keyboard input to a selected window.

Through the ioctl-call mechanism, windows can be manipulated dynamically under user-program control. Only the super-user has control access to any window in the system, otherwise a window can be controlled (i.e. modified) only by a process in its process-group or by a process with write permission on the window.

7.4.1 Basic Window Attributes

In the SM90 UNIX window system, windows occupy independent and changeable rectangles of raster-screen surface that may overlap each other. From the system's viewpoint, windows belong to a flat space where the status of any one window is essentially independent of the status of any other window. Should a user program, however, wish to impose a hierarchy on a set of windows, where, for example, the size, location, or exposure of some windows are tied together, the ioctl-call mechanism provides the ability to do so.

For **each** raster-display, there is always a current keyboard window to which keyboard input from the user is sent. The current keyboard window can be selected under program control via a ioctl-call or via a special key combination on the keyboard. For each window, the system will keep track of what other windows are currently covering it. In addition, if a window's save-image user-flag is enabled, the system will maintain an ascii image of the window's current contents. The system can, therefore, automatically-expose and refresh the ascii-image of a window that becomes uncovered due to the erasure of all covering windows. While auto-exposure is, by default, enabled, the user can turn it off via an ioctl-call. If a window's save-image user-flag is enabled, then ascii output to the window will be placed in a save-image buffer whether or not the window is exposed. Any process writing to the window, therefore, will not be stopped if the window is covered, and the window's latest contents will be displayed when the window becomes exposed again.

For font management, each window in the system has an independent and changeable multi-font map which associates slot numbers with loaded fonts (in the SM90 UNIX implementation, there are eight slots/fonts per window). Any particular font is loaded only once by the system, even if several windows load it independently and into different slot positions.

7.5 Window Signals

Two signals are specifically related to window usage, SIGWIND and SIGMOUS. These signals are, by default, ignored. Explicit signal calls, therefore, must be issued by a user-program to enable them. When enabled, either of these signals, if they occur, will be sent to the process(es) in the applicable window's process-group.

For any particular window, the SIGWIND signal is sent whenever the window changes state: (1) becomes exposed, (2) becomes covered, (3) becomes the current keyboard window, or (4) is no longer the current keyboard window. In addition, the SIGWIND signal is sent if the dimensions or location of the window change while the window is exposed. If a window changes state under program control (i.e. via a ioctl call to the window) by a member of the window's own process group, the SIGWIND signal will not be sent since it would represent redundant information.

In the SM90 UNIX window system, the mouse device is always attached to the current keyboard window. Processes in the current keyboard window's process group can read the current mouse state (x/y coordinates and buttons) by issuing the read-mouse ioctl-call. The SIGMOUS signal is sent to the process(es) in the current keyboard window's process-group whenever the mouse changes state (is moved or a button is clicked) provided the mouse's state has been read since the last SIGMOUS signal was issued. The SM90 UNIX mouse driver does allow the super-user to directly open the mouse device and therby obtain exclusive access to the mouse. If there is an exclusive mouse owner process, all read-mouse ioctl-calls will hang until the exclusive mouse owner process closes the mouse.

7.6 Window Graphics

If a SM90 UNIX user-program wants to paint a window directly (i.e write characters and/or vectors directly into the raster bit-map) a set of library wgraphl (window graphics) routines is available for this purpose. Essentially, the wgraphl package restricts a user-program to painting within a selected window by clipping vectors and font-characters at the window boundaries. In addition, the wgraphl package handles the SIGWIND signal in a default manner: (1) puts the user-program to sleep when the window becomes covered, and (2) calls a user-defined refresh routine when the window

becomes visible again or changes size or location.

7.7 User Window Manager

In order to assist the SM90 UNIX window system user in creating new windows or manipulating already existing ones, a window-manager program has been written. This program runs as an ordinary user-process and makes use of the mouse device to allow the user to select functions from a set of displayed menus. The window-manager simply issues the appropriate window ioctl-calls to accomplish the window functions selected by the user.

Current functions that the window manager supports include:

(1) the creation of new windows.

(2) modification of windows

(3) exposure or erasure of windows

(4) selection of the current keyboard window

(5) the loading, setting, or clearing of fonts from windows.

The window manager user interface will be described in more detail when the system is actually working.

7.8 Design Overview

As mentioned in the Introduction, the SM90 UNIX window system is implemented via a set of device-driver routines in the UNIX Version-7 kernel. These routines are contained in a set of files within the UNIX system source device directory. Within the kernel there are tables describing the bitmap devices that exist (there can be several raster devices on either the NU machine or the SM90) and the windows that are currently known in each one. Associated with each window are tables that describe the fonts loaded, associated teletype drivers and keyboards, etc.

From an applications programming viewpoint, the window system is quite like the file system. In order for a program to interact with a window labeled "foo", for example, the program simply opens the file "/dev/wdev/foo" and can then use write, ioctl, and etc system-calls with respect to the file-descriptor obtained from the open call. All files in the "window" directory are simply minor devices associated with the window system.

The routines that implement the window device abstraction itself are largely

hardware-independent and should be portable across similar machine implementations. These routines interface with the UNIX teletype driver, the file-system, the signal mechanism, and contain the code which services window ioctl-calls. Most of the SM90 UNIX window system code is contained in these routines.

Three "real" devices are utilized by the SM90 UNIX window system: the raster-scan display device, the keyboard device, and the mouse device. The hardware-dependent driver code for these devices is contained in separate files from the window code mentioned above. Since the code in these files is specific to the NU-Machine hardware, it will require modification across different machine implementations. The interface between these routines and the window device code, however, is fairly simple and straighforward.

7.9 Summary

In trying to keep with the UNIX philosophy of placing minimal-but-general functionality within the kernel, the SM90 UNIX window system provides a basic window management mechanism that:

(1) is transparent to the vast majority of UNIX user programs,

(2) provides a clean user interface through the standard **ioctl**-call mechanism without the addition of any new UNIX system calls, and

(3) allows user processes to manage their windows independently and with minimal kernel imposed limitations.

A typical NUnix window layout is shown in the picture on the following page. This picture, which was produced directly from a NU raster display, shows five windows: (1) a window running the user window-manager program which displays menus to assist the user in the creation and manipulation of windows in the system, (2) a local window running the Unix shell and using two different fonts, (3) a remote window being used to connect to an 11/70 Unix system, (4) a web window running a program which generates web-like pictures, and (5) a display window running a program for the viewing of a document.

7.9.1 Elaboration of Window Ioctl Calls

In the following appendix we give the details of all the window system IOCTL calls (to the NUnix kernel).

Appendix I - Window **Ioctl** Calls

User program creation and control of windows in the SM90 UNIX window system is accomplished via a set of ioctl-calls. These calls reference structures defined in the include file <wio.h> a copy of which is attached to this appendix. As with all other ioctl-calls, a return value of -1 indicates an error, where the global symbol errno contains an error code defined in <errno.h>.

Following is a list of the ioctl calls for manipulating windows. The **WIOC**xxxx names are defined in <ioctl.h> which is included indirectly through <wio.h> via <sgtty.h>. The target window of an ioctl-call is always specified by the wfd file-descriptor argument.

ioctl(wfd, WIOC**MAKE**, (struct wdata *)wdp) —**create a window**

Make a window with label wdp->w_label. The file-descriptor of the new window is returned in wdp->w_fdes. This call must be made to an already existing window (i.e. wfd must be a window file descriptor, typically 1-[stdout]). The process-group of the window is set to the process-group of the maker process. If a window already exists with label wdp->w_label, an error code will be returned.

NOTE: As mentioned in the Overview, newly created windows are entered as character-special files (devices) in the file system directory /dev/wdev with names specified by their labels. When the last reference (i.e. opened file descriptor) to a window is closed, the window's entry is deleted from the /dev/wdev directory.

ioctl(wfd, WIOC**INIT**, (struct wdata *)wdp) —**initialize a window**

Initialize the target window according to data contained in the wdata structure pointed to by wdp. A window can only be initialized once, and initialization must preceed all other actions regarding the window (i.e. all other window ioctl-calls are applicable only after initialization has occured).

As defined in <wio.h>, user-settable window attributes include: (1) user-flag parameter wdp->w_uflags, (2) upper-left-hand coordinates wdp->w_ulcx and wdp->w_ulcy in pixels, (3) size parameters wdp->w_width and wdp->w_height in pixels, (4) paper-color parameter wdp->w_color, (5) draw-mode parameter wdp->w_mode, (6) character-color parameter wdp->w_ccolor, and (7) ascii-save buffer size parameter wdp->w_assize which is applicable only if the SAVEWI bit is set in wdp->w_uflags.

The user-settable flag bits in wdp->w_uflags define specific operational characteristics of the target window including: H19 escape sequence emulation (H19 bit), scrolling (SCROLL bit) or wrapping, cursor suppression (CSROFF bit), line-spacing based on the biggest loaded font (BFLSP bit) or the current font, label display suppression (NLABEL bit), and ascii-save image restoration (SAVEWI bit).

NOTE: The WIOCINIT call is maintained for historical purposes in the NU-

Machine implementation. Ideally, window initialization should be merged into the WIOCMAKE ioctl-call.

NOTE: All ioctl-calls following WIOCMAKE/WIOCINIT manipulate existing windows. From an applications programming viewpoint, in order for a program to interact with a window labeled "foo", for example, the program can simply open the file "/dev/wdev/foo" and address write, ioctl, etc. system-calls to the file-descriptor obtained from the open call.

ioctl(wfd, WIOC**DRAW**, 0) — **draw a window**

Draw the target window. If the target window is partially or totally hidden from view, it will be popped to the viewing surface.

ioctl(wfd, WIOC**ERAS**, 0) — **erase a window**

Erase the target window. If the target window is fully visible, its image will be erased. In addition, if raster auto-exposure has not been disabled (see WIOCSRSD call below), all windows that were covered by the erased window exclusively will be automatically popped to the viewing surface.

ioctl(wfd, WIOC**GETD**, (struct wdata *)wdp) — **get window attributes**

Fill in the wdata structure pointed to by wdp for the target window. This call returns essentially all the relevant data about the target window including its label, available fonts, system and user status, position and size, and major/minor device number. See also the window calls that return a subset of this info: WIOCLABL, WIOCSTAT, and WIOCRACD.

ioctl(wfd, WIOC**SETD**, (struct wdata *)wdp) — **set window attributes**

Modify the target window according to data contained in the wdata structure pointed to by wdp. All user-settable data for the target window is affected via this call (refer to the WIOCINIT call above for a list of settable attributes).

ioctl(wfd, WIOC**LFNT**, (struct wfdata *)wfp) — **load a font into a window**

Load the font specified by wfp->wf_name into slot wfp->wf_slot in the target window. If a font is already loaded in wfp->wf_slot it will be cleared before wfp->wf_name is loaded. If the given font name does not refer (via search rules) to a valid font file, an error code will be returned.

ioctl(wfd, WIOC**SFNT**, (struct wfdata *)wfp) — **set current font for a window**

Set the current font in the target window to the font specified by the ordinal passed in via wfp->wf_slot. If no font is loaded in this slot, an error code will be returned.

ioctl(wfd, WIOC**CFNT**, (struct wfdata *)wfp) — **clear a font wrt a window**

Clear the font specified by slot wfp->wf_slot in the target window. If no font is loaded in wfp->wf_slot, the clear command will be ignored.

ioctl(wfd, WIOC**GFNT**, (struct wfdata *)wfp) — **get font attributes**

Fill in the wfdata structure pointed to by wfp for the font loaded in wfp->wf_slot in the target window. This call returns essentially all the relevant data about the selected font, including its buffer address in memory, size, and effective character dimensions.

ioctl(wfd, WIOC**RDMS**, (long *)mdata) — **read the mouse**

Read the target window's mouse, current mouse data is returned in mdata. This call returns if(when) the target window is(becomes) the current keyboard window. The contents of the mouse data register mdata is described in an attachment to this appendix.

ioctl(wfd, WIOC**GRSD**, (struct rsdata *)rdp) — **get raster device info**

Fill in the rsdata structure pointed to by rdp for the target window. This call returns essentially all the relevant data about the raster-display to which the target window belongs, including its base-address in memory, size, line-length, and dimensions.

ioctl(wfd, WIOC**STAT**, (struct wstat *)wsp) — **get system and user status**

Return the system and user status flags for the target window in wsp->w_flags and wsp->w_uflags respectively. This call is intended primarily for programs that do not need all the data obtainable via the WIOCGETD call.

ioctl(wfd, WIOC**LABL**, (char *)w_label) — **get window label**

Return the label of the target window in w_label. Labels are simply ascii strings used to "name" a window. Normally but not always, the name of a window appears visually as part of the window. This call is intended primarily for programs that do not need all the data obtainable via the WIOCGETD call.

ioctl(wfd, WIOC**RACD**, (struct wracd *)wrp) — **get row/col window info**

Return the number of rows and columns (in the current font) of the target window in wrp->w_nrow and wrp->w_ncol respectively. This call is intended primarily for programs that do not need all the data obtainable via the WIOCGETD call.

ioctl(wfd, WIOC**SKBD**, 0) — **establish current keyboard**

Make the target window the current keyboard window. If the target window is partially or totally hidden from view, it will be popped to the viewing surface. This call provides the ability to set the current keyboard window under program control (i.e. verses the keyboard SYSTEM-key mechanism).

NOTE: If two or more processes try setting the current keyboard window simultaneously, a conflict will result. At present, the system does not support the ability to lock the current keyboard window against usurpation.

ioctl(wfd, WIOC**SRSD**, (short)r_uflags) — **set raster dev info**

Set the raster-scan-display user flags for the raster to which the target window belongs to the value specified by r_uflags. Currently, only one flag is settable: the NOAUTOX flag specifies that auto-window exposure is to be turned off in the target raster display.

7.9.2 Window System Design Overview

Appendix II - Design Overview

The UNIX window system for the SM90 is implemented via a set of device-driver routines in the UNIX Version-7 kernel. These routines are contained in a set of files within the UNIX system source device directory.

7.9.2.1 **Window** Device Code

The routines that implement the window device abstraction itself are largely hardware-independent and should be portable across similar machine implementations. Indeed, they have already been ported from the NU machine to the Sm90. These routines interface with the UNIX teletype driver, the file-system, the signal mechanism, and contain the code which services window ioctl-calls. Most of the window system code is contained in these routines which are contained in three files:

dev/**wd.c** - contains routines which service calls from the UNIX character device switch cdevsw for window devices. Such calls include the opening, closing, reading, writing, and ioctl of windows.

dev/**wfont.c** - contains routines which handle font manipulation for windows including the loading, setting, and clearing of fonts.

dev/**window.c** - contains routines which provide the basic window function library including making, initializing, drawing, erasing, moving, scrolling, etc. of windows.

Essentially all the code in these three files should be portable across machine implementations. One exception is the routine checkfont() in dev/wfont.c which is dependent upon the particular font format used by the window system. Depending upon the font format chosen, this routine should be adapted to it. On the SM90 and on the NU machine the .ft font format is used, a description of which is attached to this appendix. Each font loaded by the window system is characterized by a set of effective dimensions which define the size of a "standard" character. These dimensions along with the maximum character dimensions of the font are determined by the checkfont() routine. For .ft fonts, the effective font character size is defined by the dimensions of the space character.

Another window system code dependence concerns the pixel coordinate

(0,0) which, on the NU-Machine and on the Numelec hardware, (the SM90 bitmap) refers to the upper-left-hand corner of the raster-display. For machines where (0,0) refers to the lower-left-hand corner of the raster-display, references in the code to w_ulcy (window upper-left-corner **y**) will have to be changed appropriately.

7.9.2.2 **Hardware** Device Code

Three hardware devices are utilized by the SM90 UNIX window system: the raster-scan display device, the keyboard device, and the mouse device. The hardware-dependent routines which drive these devices are contained in separate files from the window device code mentioned above. These files are:

dev/**rsd.c** dev/**rsda.a68** - contain routines which manage the raster-display and implement basic raster-op functions. A substantial part of the code in dev/rsd.c is hardware-independent and should be portable across machines. The code which actually paints the raster-display is contained in dev/rsda.a68.

dev/**pci.c** dev/**keybd.c** - contain routines for the servicing and routing of keyboard input. The Micro-Switch keyboard used by the NU-Machine interrupts to a programmable communications interface dev/pci.c which forwards data to the keyboard's line-driver dev/keybd.c for translation to ascii and multiplexing to the appropriate window teletype driver.

dev/**mouse.c** - contains routines which are responsible for the initializing, reading, and interrupt-servicing of the mouse device. These routines are not essential to the window system and are applicable only if a mouse-device is to be used with the raster-display.

Since most of the code in these files is specific to the NU-Machine hardware, it will require modification across different machine implementations. The interface between these routines and the window device code, however, is fairly simple and straighforward.

7.9.3 Bootstrap Procedure

The file conf/wrc.c contains the cdevsw table for NUnix. By comparing this table with the listing of the NUnix /dev directory contained in the file conf/devlist, it is apparent that /dev/console refers to a window device, namely the raster-window. When /dev/console is opened by init, a full-screen raster window is automatically made by the wdopen() routine contained in dev/wd.c. In this way, when NUnix is booted, a raster-window is made, initialized to full-screen dimensions, and the shell process run in it. Similarly, when coming up multi-user, the entry /dev/WRUNIX is placed in the /etc/ttys file, and also refers to the raster-window. In addition, since all sub-windows are placed in the directory /dev/wdev by the kernel, this directory should be cleaned out when coming up multi-user by placing a "rm /dev/wdev/*" command in the /etc/rc file.

7.10 Format of Font Files

Appendix III - Format of .ft Font Files

Both the SM90 and MIT's NU machines use the same format for font files, which begin with an array of FONTSIZE (128) **fcdefs** followed by aggregated mini-raster data. A program may read the initial portion of a .ft file to get per-character summary data, or may read the entire file in a single operation to load a complete font.

The C structure that describes each font character is:

```
struct fcdef                /* font character definition    */
{
        char    fc_hs;      /* horizontal size in bits      */
        char    fc_vs;      /* vertical size                */
        char    fc_ha;      /* horizontal adjust (signed)   */
        char    fc_va;      /* vertical adjust (signed)     */
        char    fc_hi;      /* horizontal increment         */
        char    fc_vi;      /* vertical increment           */
        short   fc_mr;      /* relative mini-raster pointer */
};

define rasterptr(fcp) (((char *)&((fcp)->fc_mr))+((fcp)->fc_mr))
define rastersize(fcp) (((fcp)->fc_vs)*((((fcp)->fc_hs)+15)>>4))
```

Given (struct fcdef *)fcp,

(short *)rasterptr(fcp) = a pointer to the character's mini-raster.
(short)rastersize(fcp) = the size of the mini-raster in bytes.

Each mini-raster is dealt with as 16-bit words; hence it must be word-aligned, and consist of fc_hs raster lines each of which contains an integral number of 16-bit data words. The actual position of upper-left corner of miniraster is (curx+fc_ha, cury+fc_va).

Every word of mini-raster information is stored HIGH byte first, ala MC-68000. Low order bit of first word is left-most raster point. Bit-0 of first word thus corresponds to the upper-left corner of the character.

The actual bit pattern of a character is flush left in its mini-raster. The bits to the right of the pattern (i.e. to the right of fc_hs) and before the short boundary must be 0. Normally, fc_va is negative, thus implying that coordinate (0, 0) is upper left.

The space character of the font (' ') defines the "effective" character size for the font as a whole. Within the window system, scrolling and other window attributes can also come from the biggest font loaded, rather than the font being used at the moment.

Example of .ft Character

(The typeset version of this is very badly presented just now, due to the proportional spacing, etc, done by the versatek. Sorry).

```
ASCII CHAR FC_HS FC_VS  FC_HA  C_VA  FC_HI  FC_VI  MRSIZE
 0x61  'a'   10   25     1    -18    12     0      50

                    .
                    .
 (curx+fc_ha,cury+fc_va)+--------
                    .--------
                    .--------
                    .--------
                    .--------
                    .--------
                    .--------
                    .--xxxxx---
                    .-xxxxxxx--
                    .-xx---xxx-
                    .------xx-
                    .----xxxxx-
                    .-xxxxxxxx-
                    .xxx----xx-
                    .xx-----xx-
                    .xx----xxx-
                    .xxxxxxxxxx
                    .-xxxxx-xxx
baseline . . (curx,cury)+---------.+(curx+fc_hi,cury+fc_vi) . . .
                    .--------
                    .--------
                    .--------
                    .--------
                    .--------
                    .--------
                    .
```

In the SM90 window system, users need not be concerned with these formats very much; either the window system is used, **or** a SET FONT command is provided, in which case all of this is "looked after" behind the scenes.

The format shown above is the same as what FE builds as an **icon** or "Char Def". When you compile an "icon file" that FE builds, it's exactly the same thing as having a C variable initialized to contain the above structure and values.

CHAPTER 8

The Bitmap Terminal Emmulator

In Retrospect: The "terminal emmulator" (TESM) that we got from MIT served us well in debugging the Raster-OPs on the Numelec. The program is also useful for producing screen fulls of text that use different fonts, special video effects, and etc.

Most of the credit for this goodie goes to MIT; it was a life savor for me. I worked extensively with the escape sequences described herein, and added some goodies to TE so that it can be used to produce videotex-like screen-fulls. See the DEMO section in BITMAP paper for more details.

The rest of this chapter contains the TE document verbatim.

The Bitmap Terminal Emmulator

(draft version)

Kevin Pammett

I.N.R.I.A.
Rocquencourt, France.

8.1 OVERVIEW

A new terminal emulator is now available as tesm.r68. Tesm (terminal emulator and screen manager) simulates a Heath terminal. It also knows about some .scan format sequences and graphics commands.

Tesm writes directly on to video memory. It can have multiple fonts, highlighted characters, and a number of other features not available from con.

The keyboard is read in alternate mode, and keys may be reassigned.

When tesm comes up, it reads in the file .tesm.init from your connected directory. Characters in that file are passed to the screen handler. Tesm also has an offline mode where characters are directly passed to the screen handler.

The following document describes TE. It is the "help" text supplied by MIT on

their NUnix distribution tape.

8.2 DESCRIPTION

TESM (terminal emulator and screen manager) is a telnet program that read and writes on to a host through /dev/tty1. It does direct writes on the video screen and thus can show different fonts and draw vectors. Tesm handles the AI keyboard in alternate mode and by interrupts. It thus can assign any code to any of the keys. 5 family of control keys may be defined. Also, each key may be operated upon by a function that does useful things to it (e.g., prepend 033 etc.). Tesm also has an offline mode (like the Heath terminal) which passes keys typed directly to the screen handler. In this way internal variables may be set. Tesm can also read in a file and execute its contents (i.e. pass it to the screen handler).

Tesm simulates a Heath terminal. It also knows about some of the codes of .scan format. A number of graphics sequences such as raster operations and fill operations are also provided.

Since tesm does output in RAW mode, it can manually send the ^S and ^Q characters. This is a great win for cretinous systems like Twenex.

When tesm comes up, it tries to read the file <connect-directory>/.tesm.init; upon which the contents are executed.

8.2.1 Startup Defaults

The defaults upon starting up TE are as follows:

All keys are set to the default Unix keys. However the following are different:

TERMINAL	get into offline mode when down
ALT LOCK	same as TERMINAL
MODE LOCK	puts tesm in font 0,restores default key binding
CALL	^C; in offline mode, causes tesm to terminate
HOLD OUTPUT	freeze the screen
RESUME	unfreeze the screen
Thumbs and Fingers	associated Heath escape codes
META	preceed character by ascii 033. In offline mode, set the 8th bit in the character.
load font /Fonts/CRT/nunix.ft as font 0:	^]L0/Fonts/CRT/nunix.ft^@
font 0:	$f0
black background:	$q
clear screen:	$E
character write mode:	$S0
do not insert chars:	$O
do not fill over chars:	^]O0
fill mode 0:	^]M0
fill pattern 0:	^]P0

8.3 Escape Sequences

The following escape sequences are defined (<n> is an 8 bit integer, <N> is a 16 bit signed integer with high byte first, <c> normally means an 8 bit character with code >= 32):

8.3.1 Heath Escape Sequences

The emmulator provides most Heath escape sequences includeing Insert/Delete line and Insert/Delete characters. Also:

^J	Newline; when cursor come to end of screen, you can either wrap, or scroll. See ^]E
^M	carraige return
^H	backspace
^I	tab
^L	erase screen
^G	bell
^C	if offline, quit
$A	cursor up one line
$B	cursor down one line
$C	cursor right one line
$D	cursor left one line
$H	home cursor
$J	erase to end of screen
$K	erase to end of line
$Y<cy><cx>	position cursor to y = <cy> - 32, x = <cx> - 32
$E	same as ^L
$b	erase beginning of screen
$l	erase line
$o	erase beginning of line
$L	insert line
$M	delete line
$N	delete character
$@	enter insert character mode
$O	exit insert character mode
$z	Reset
$p	Reverse video character, black on white background
$q	Normal video character, white on black background
$<	Enter ANSI mode

8.3.2 ANSI compatible modes

Here are the ANSI compatible modes:

$[?5h	reverse video
$[?5l	normal video
$[?2h	Enter Heath mode

8.3.3 .Scan Format Escapes

Here we describe the .scan format escape sequences that are provided.

$f<n> Switch to font <n>

$m<X><Y> position cursor absolutely at (X, Y).
(0, 0) is upper left corner of the screen.

$v<dX><dY> draw vector from current cursor to relative positon (dX, dY).

$s<d1><d2>... draw short vectors; each byte consits of a 4 left bits of signed dx and 4 right bits of signed dy.
If dx = -8, then if dy =
-8terminate short vector sequence
0ignore the next short vector.

$^L Reset terminal. Fonts remain.

$S<n> Character writing mode is set to 2 low order bits of n:
0 store character
1 set (or) character
2 clear (and complement of character)
3 xor character

8.3.4 Graphic and TE-Specific Escapes

The following graphic sequences and TESM specific sequences are defined. The escape character here is ^] (octal code 035):

$return to debugger
$} quit with a core dump, abort

^]L<n><font name>^@Load a font. The font number is n,
<font name> is a complete path name for the font in .ft format. The name is terminated with ^@ (break character).
Maximum of 8 fonts are currently allowed.

^]X<file name>^@ Execute the contents of the file specified by <file name>. This will put the terminal in offline mode and read the file as if it was typed at the terminal.

^]K<k><c0><c1><s0><s1>Set key <k> to unshifted code <c0> and <c1> and shifted code <s0>, <s1>. The Key Table in tesm.c lists the corresponding keyboard key to key number <k> assignments
For a description of codes <c0>, <c1>, <s0>, and <s1>, refer to tesm.c.

^]R<n><Sx><Sy><Dx><Dy><W><H>Perform raster operation of source

rectangle at (Sx, Sy) on destination rectangle at (Dx, Dy). Size of the rectangle is <W> wide and <H> high. <n> is the operation code:
0 store
1 set
2 clear
3 xor

^]F<n><X><Y><W><H><p0><p1><p2><p3> Fill rectangle at (X, Y) and size (W, H) according to mode <n>:
0 store
1 set
2 clear
3 xor
4 complement rectangle (pattern ignored)
and according to pattern p0, ..., p3. The 4 pattern bytes specify a 4 x 8 rectangle which will be replicated in the rectangle to be filled.

^]O<c> Fill over a character after it is written according to fill mode and fill pattern specified below.
<c> = 1: start filling over, <c> = 0: stop filling over.
This is the way to obtain highlighted or halftone characters.
This command is reset when a $p or $q occurs.

^]M<c> Set fill mode according to <c>:
0 store
1 set
2 clear
3 xor
4 complement character

^]P<c> Select one of the 8 built in patterns for filling over chars according to <c>.

^]|<comment>^@ <comment> is any sequence of characters terminate by ^@ which will be ignored.

^]m<c> set the following modes depending on <c>:
0 overstrike characters
o do not overstrike characters
S scroll when cursor at end of screen
s do not scroll, wrap
C clear newline upon receipt of ^J
c do not clear newline

8.4 Notes on TESM efficiency

TESM is at least as efficient as CON. Apparently the bottleneck is due to UNIX I/O, so not much more improvement can be expected. Though TESM hacks the screen directly, it runs as fast as CON does.

There are 2 modes which have a direct influence on the speed of TESM.

1- Character printing mode
2- screen wrap mode

1- A character sent to TESM may be printed in a variety of modes

as specified by any one of the following sequences:

$@	insert character mode: will slow tesm
$p	character reverse video: will slow tesm
^]0	character highlight: will slow tesm

Thus to speed up tesm, make sure none of the above is used.

A further speed up of 20% may be obtained if the chracter write mode is other than 'storé. I.e. if

$S	is other than 0, things will be faster.

The problem here is that TESM will be in character overstrike mode, and hence probably not usable with a text editor. However, this mode is great for making listings, etc.

2- Normally, TESM wraps the cursor around the screen when it gets to the bottom of the screen. If you don't like this, you can get into scroll mode so that TESM will behave like an H19. However, this mode is slower since a lot of bits have to be copied on the screen.

^]mS	puts you into scroll mode
^]ms	puts you into wrap mode

N.B.:Initially, TESM does not clear a newline upon receipt of a ^J. However, once the cursor has wrapped around the screen, it begins to clear newlines. This is necessary since some text editors use ^J to move the cursor down one line.

8.5 Further Information

It appears that TE will be supported and distributed our of Sophia via (perhaps indirectly) Gilles Kahn. Their version handles chinese and arabic fonts and is generally more stable.

CHAPTER 9

A Command-Language Iteration Primitive – Revisited

In Retrospect: ITER is my second generation prototype for a command iteration primitive. The first was for VAX/VMS; the second, a complete rewrite, is for UNIX. In the latter I didn't get to play so much with visual effects (no VT100s around), but the iterator written in C is much more powerful and flexible than the DCL version I first did for VAX/VMS.

Someday iteration will receive just treatment within the command languages of current operating systems. Just now, unfortunately, this is not the case; hence this ITERator.

This version of ITER has been distributed to Canada, the US, and Europe, over the UNIX uucp network. Perhaps my next iterator will be a real one - written in Ada and running in a multi-window bitmap system...

The rest of this chapter contains the ITER document verbatim.

ITER – A UNIX Tool For Command Iteration

(Tutorial)

Kevin Pammett

I.N.R.I.A.
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay Cedex
France

Abstract

The notion of **iteration** itself is not new, but being able to "iterate a command over a set of operands" at the (UNIX) command level is. What this means is that you get to specify a perfectly arbitrary UNIX command, and the ITERator will execute this command once for each operand (string) that is found in a set of elements that you also specify.

The most important aspect of ITER is that it is highly interactive. At any point you can interrupt the iteration, regain control, and interactively perform a whole host of operations; you never loose control of the operation, and have maximum flexibility with respect to manipulating the iteration environment.

9.1 NAME

iter - execute a single UNIX command over a **set** of operands.

9.2 SYNOPSIS

ITER [-qvfe] cmd_word_1 ...
or
ITER -s[qvfe] set_name cmd_word_1 ...

Or, you can invoke the iterator with its "french name", **IT**, in which case all user dialogue will be in french. ITER and IT are the same program.

Note: in this document I use the nomenclature **ITER** *and* **IT** *to refer to the command iterator. On the UNIX system distributed by INRIA, the actual name that you type is 'iter' or 'it' (in lower case).*

9.3 DESCRIPTION

ITER is a **command iterator**. That is to say, ITER is a UNIX utility whose primary function is to issue commands to the shell. Normally, a single "skeleton" or "template" command is specified along with a set of arguments, and ITER issues this command several times - once for each element in the set.

A **set** is a collection of arguments (lines) read from a specified file or (in default) read from standard input.

A concrete example:

Suppose you have a file called 'body' which contains the names of several C program files. You could produce the associated object files by hand - that is, by repeatedly invoking the 'cc' command with the right option flags, each time giving this command with a different C program file name. If needed, you might consult the 'body' file (or LS) to remind you of the names.

On the other hand, the iteration:

$ iter -s body cc -c

would do all that work for you. If you gave '-**sv**' (instead of '-s') you could watch ITER do its work, and if you gave '-sv**q**' you could initiate a dialogue to selectively choose which files you wanted to compile.

Using ITER terminology, the command **template** is "cc -c", and the Name-SET (set) is the file called 'body'. Since ITER normally reads sets from standard input, the **-s** flag is necessary to identify 'body' as the set name and not the first

word of the command template.

I use the term **Name-SET** simply to draw attention to the fact that I do indeed mean something quite specific by it. In general, the notion of "a set" is quite intuitive and this is more or less what Name-SETs are - but see the more complete discussion of Name-SETs which appears near the end of this document.

9.3.1 Fundamental Operation

In general, the operation of ITER is to take each element from a set, successively, and to use this in conjunction with a template command to perform a substitution operation. Each time a new command is formed, it is passed on to the UNIX shell for execution. In default, an iteration is executed entirely from beginning to end without user intervention. At any point, however, the interactive user can interrupt an iteration and retain control over the operation. At this point, ("query mode") there are a host of options available to manipulate the commands, the operands, or the iteration itself.

In effect, there are almost two distinct ways of interacting with ITER. When you invoke it from the command line, the syntax shown in SYNOPSIS applies and, if all goes well, there is nothing more to it. An entire iteration can be executed without any user interaction, so ITER is appropriate for command-file usage as well as in background iterative processing.

On the other hand, ITER appears more like a sub-system when you enter query mode. The interactive dialogue that is engaged in query mode is a language unto itself, in the same sense that this is the case while you are "in" an editor. An "escape" is provided so that you can enter an ordinary UNIX command while in query mode, but the assumption is that query mode is a specialized form of user interaction for the purpose of manipulating a suspended iteration.

You can invoke ITER and enter query mode directly by using the **-Q** command line option. Query mode is also extered on interrupt (DEL, or the ^C terminal sequence) or if any command issued by ITER terminates in **error**. In this way, the interactive user is guaranteed that if something does go wrong, he'll be able to intervene and fix the problem before anything further is attempted and without losing any of the iteration context in the meanwhile.

An iteration is always invoked by entering a single ITER command, either as part of a pipeline or as a simple (ordinary) UNIX command. Accordingly, a shell command file can invoke "planned" iterations, or, even, a C program can "call the iterator" (via the SYSTEM library function).

9.3.2 Command Line Options

If you want to use one or more command-line options, they must all be given as a single argument to ITER with the customary '-' prefix to signal that this string is to be interpretted as options flags. In the typical UNIX fashion, unrecognized flag characters are ignored without any message being produced. The flags can be given in any order and in either upper or lower case.

In all cases, the following command-line options apply:

-V -Verbose mode.

The **-V** option causes ITER to be a lot more chatty than it would otherwise be. For example, -V causes each ITER command to be displayed just before it is passed off to a subprocess shell. In query mode, there are numerous informational messages that remind you of what was done as the result of a command you entered. Being in TERSE mode causes these messages to be suppressed. The -V option flag turns on "verbose mode", the default for which is TERSE. If you give several -Vs on the command line, you will still come up in VERBOSE mode, no matter how many Vs there are. Note that this is somewhat different from the query mode command, 'V', which toggles this mode back and forth between TERSE and VERBOSE.

There is no command line option to select TERSE mode.

-E -Use English as the dialogue language.

-F -Use French as the dialogue language.

As an experiment, ITER is currently prepared to speak either French or English. In fact, ITER goes by two names if it is properly installed on your UNIX system. Using the name ITER causes the iterator to come up in English. If you invoke it by its alternate name, **IT**, the default is for the iterator to speak French. IT and ITER are the same program; one is simply a file-system LiNk to the other.

In either case, the **-F** and **-E** (for "French" and "English") options cause IT or ITER to choose the indicated language as the default for all user dialogue. These flags override the default which is implied by the invocation name. For example:

$ it -e echo

will cause the iterator to speak english and read the Name-SET from standard input. This example has almost the same effect as CAT invoked with no arguments. The (trivial) difference is that ITER will not let you iterate over a set whose cardinality is 0, while CAT is perfectly happy to display an empty file.

No matter how you invoke ITER or IT, the associated prompt will be used in all message output by the tool. Just now, there is no way to change this - you get "IT: ..." if the current language is french, and "ITER: ..." in English - no matter how you invoked the iterator. If you

switch languages during an iteration, (see the 'L' query mode command), the prompt changes to match the language.

-Q -Query Mode.

The -**Q** command line option flag causes ITER to come up in query mode directly. That is, you will be queried about the disposition of each command that is synthesized, before ITER even tries to execute it. Entering query mode this way is not distinguished from doing so via ^C (or whatever you have your keyboard interrupt set to). This is also exactly the same mode which is entered when any iterated command terminates with an error exit status. You can find out the exact exit status via the '=V' query mode command.

It is this "query mode" which ITER leaves if you follow any query mode command with the UNIX-inspired "&" syntax. See the internal HELP in ITER for more details. If you are doing an interation and are not in query mode, you can always switch to query mode by interrupting ITER (via ^C) no matter what it is doing. If a subcommand was in progress, UNIX will kill that process and ITER will query you as to whether you wish to restart it.

-S name -Set (actually, Name-SET), the name of which is specified as an argument which **must** be given along with the -S.

If a -**S** flag (for "set") is given, the set is assumed to be already contained in the named file. For example:

$ iter -fsv changes touch -c &

will update the file-modified time ('touch') for all files named in the "set" (file) called 'changes'. The trailing & is nothing special; it is interpretted by the invoking shell and causes the iteration to happen in the background. For planned applications or for users who happen to have "sets" around, the -S form will be heavily used. Otherwise, ITER is normally used in a pipeline even though it will indeed let you type in the set directly if it is invoked with no -S and it is not part of a filter.

There will have to be some more qualifiers when the bitmap (multiple window) version of ITER is available.

9.3.3 The Command Template

The template that ITER uses is always given as command-line arguments to ITER. Once the command-line options are interpretted, ITER constructs the command template by concatenating the remaining command-line arguments together with a single delimiting space following each argument given.

The term "command template" is used because this is the pattern used by ITER as the basis for each new command that is executed. In the above example, the user gave "touch -c" as the command to be iterated. Since parameter substitution is fundamental to iteration, the actual command template used is

"touch -c %E"; the '**%E**' being the argument formal or "place marker" used by ITER when doing the substitution each time around the loop.

If the user does not specify an argument formal, ITER assumes a delimited '%E' at the end of the user-given command template, as in the above example.

On the other hand, if a '%e' (equivalent to '%E') character sequence appears anywhere in what the user gives, no such defaulting is done. For example,

```
$ iter -s headers 'pr %E | lpr -m' &
```

invokes an iteration to spool "the header files" in the background in such a way that the user receives mail (-m) when the printing is done. In this iteration, the command template is "pr %E | lpr -m". There is no '%E' at the end of the template, and ITER does not force any delimiters around the actual parameter when substituted. This is particularily important in an example like:

```
$ iter -svq important 'cp %e /tmp/%e.0; cat %e >>save'
```

which takes each file named in the set called 'important' and makes a /tmp copy of it in such a way that the resultant file names are the same as the "source" file names except that each has a '.0' suffix. Like an archiver, the iteration also adds each of the saved files onto the end of an "accumulator file" called 'save'. This iteration is an example of how ITER lets one make up "aggregate commands" for UNIX primitives (like CP) which don't otherwise handle such a thing.

In each of the two preceding examples, single (or double) quotes are required because the command templates contain characters that would otherwise have significance to the shell.

ITER parameter substitution happens after the shell does whatever processing it normally does on invoking ITER in the first place. Each time the substitution is done, however, no notice is taken of quote levels of any sort.

In effect, an iterated command is processed twice by the shell, and once by ITER. The command template is processed in the usual way by the shell when ITER is first invoked. Thus, for example, one could use '.$count' in place of the '.0' suffix string above, and the suffix would be taken from the shell variable either (A) each time around the loop or (B) at the beginning of the iteration, depending on whether single or double quotes were used to enter the template, respectively.

When ITER parameter substitution is done, all '%E' strings in the template are replaced by the current element from the set each time around, no matter where in the template the formal(s) appear, and irrespective of all notions of quoting. Once the "instantiated command" has been constructed by ITER, it is again subject to shell processing because the fundamental operation of ITER is to pass on commands to a subprocess shell.

9.3.4 Using ITER as a Filter

ITER can be used at the beginning, the end, or in the middle of a UNIX pipeline or filter. In fact, the expected usage of ITER is seen in this common example:

```
$ ls *.c | iter ed
```

which invokes the ED command for each "C program" in the current directory. The "set" (or "collection") of elements is read from the standard input, line by line, in exactly the same way as "set elements" are read from existing files.

Since UNIX is rich in filtering tools, arbitrary processing can be done in a pipeline to prepare a set or collection of elements for a subsequent iteration. For example:

```
$ ls *.c | grep -v r | iter cc -c
```

would make an object file ("cc -c") for each C source file in the current directory whose name does not contain the letter "r". The "source" (LS, in this case) could be anything, including something like "cat name1 name2...", which provides for Name-SET concatenation. Likewise, the middle filtering (GREP, in this case) could be any number of processes, including the stream editor (SED), EVAL, or a user-writen ESP-like utility. (ESP processes the abritrary set expression language that was prototyped on VAX/VMS).

Using ITER in the middle or at the beginning of a pipeline is also quite feasible. Whenever it is the output of an iteration that you want to "pipe" on to another process, simply redirect the output of the entire iteration, just as you would for any UNIX command.

For example:

```
$ crypt password1 <secrets | iter crypt password2 | mail friend
```

would decrypt all of my 'secrets' message files and mail them off to 'friend'. Note that both the message files and the list of their names is maintained in a CRYPTed form. Decrypting 'secrets' supplies the set for the iteration, which then decrypts each of the files to produce the associated text. The files would never exist in a decrypted form until their receipt (as mail) by the intended recipient.

The transformation part in such an iteration, ("crypt password2", above), could be any UNIX or user-written utility, including a command file. In this way, ITER can be used to generate data files for a variety of purposes.

It is not necessary to use ITER in a pipeline, though. If no Name-SET is given by any means - be it via the "-S name" syntax or by ITER being in a pipeline - ITER reads the elements, line by line, from the standard input which would be the terminal. In this way you can do an explicit iteration over a user-given set directly. But see FEEDBACK, below.

9.4 Query Mode Commands

Once ITER enters query mode via whatever means, there are a host of commands or sub commands that you can use to alter things about the iteration, find out what is going on, etc.

The assumed scenario in query mode is always the same: ITER displays the template command having already done the substitution of the current Name-SET operand, and asks you if you want to execute this "instantiated command". At this point you can answer with any of the commands documented below.

If you are actually using ITER, type '?' (or any invalid response) to the query mode prompt and ITER will produce a screen-full description of the valid responses that you can enter. This screen full is what is referred to as the "internal HELP" for ITER; this (much larger) document here is the "external HELP file" or tutorial.

The following commands are recognized by ITER in query mode. The metasyntax, "[x]", is used to signify that the X part is optional. In general, query mode commands are single characters and you CANNOT give the full (e.g. "yes") form. (This is because I am more in favor of installing a real command parser in ITER, like a YACC-generated one. See FEEDBACK (11) for details).

Y -Yes (or '0' for Oui). The indicated command is passed on to a child process via SYSTEM for execution. This should have the same semantics as executing it at command level. If the command does not return an error status, an automatic "advance 1" is done so that afterwards, ITER proposes the subsequent command for execution.

If you are in query mode and the end of the Name-SET is reached, ITER produces a message to this effect and prompts for what you'd like to do. In "non-query" mode, ITER simply returns to the calling shell when the end of the Name-SET is reached. See FEEDBACK for a discussion of this issue.

N -For No or Next. An "advance 1" is done, which, for now, has the effect of moving you ahead in the Name-SET by exactly one element. There is no way to automatically go backwards thru the set, skip according to a "match criteria like /xxx/", or etc. See FEEDBACK for more on this.

!<cmd>-To escape to the UNIX command level and execute the indicated command. This string is not subjected to the same "white space removal" and trailing-'&' syntax as are the other query mode commands. It is, however, searched for the usual ITER formal parameter marker syntax (%e or %E) and is therefore subject to the same actual parameter substitution as the ITER command template is. If no formal marker is found in this string, NO default is supplied.

Once you give an exclamation mark command, ITER remembers its template form and will display that back to you if you consult '=V'. A subsequent exclamation point command in which you type **only** the

exclamation character is taken to mean that you want to reexecute this same "shell" command. Note that when you do this, the command template is resubstituted with the current Name-SET element, thus resulting in a different command under normal circumstances. Note also that this command template is completely independent of the ITER command template.

x -For eXit. This is the expected way for one to leave an iteration. A cleanup is done and ITER simply returns to its caller with a non-error status code. If you are in a nested iteration, control will return to the ITER which is "above" this one in the invocation heirarchy. That is, 'X' takes you up one level in a nested iteration. All context for the inner ITER is lost - there is no way to go back and forth between nested ITERs. We have to wait for multi-screens and multi-process windows for that kind of stuff.

See FEEDBACK (12) for a discussion of why we don't really have an 'e' (for 'exit') command.

q -Quit. The iterator returns a special exit status code which is considered an error by the shell but which is special-cased by ITER. The effect is to let you exit from a nested iteration no matter how many levels deep you were. The entire ITER process tree is discarded and you return to the calling shell. See FEEDBACK (12) for a discussion of the naming of the 'x' versus 'q' commands.

c -For 'Change'. This command lets you alter the ITER command template that you gave when you first invoked ITER. Rather than simply forcing you to retype the command template, I'd like to experiment with a way to let you edit it, even though we don't really have the screen-based, tools (a specialized 1-line editor) that I'd like to have for this...

What actually happens is that the current ITER command template is written into a 1-line /tmp file. Then, a subprocess is invoked with /bin/ed in it and a buffer which is pre-loaded with this /tmp file. If you print (via '1,$P') this buffer, you see the current command template - including the '%E' formal parameter markers. You are expected to use ED in any way whatsoever to change the template, and you have to realize that you must write out the file and quit ED to return to ITER. When you leave the editor, the first line of the /tmp file is read and used to reload the command template. There is no %E-defaulting at this point; how ever you leave the command template is exactly as ITER will try to use it. See FEEDBACK for some ideas on "getting your own editor", etc.

l[x] -For 'Language' switching. The Xs accepted are 'F' for French, or 'A' (anglais) or 'E' for English.

This command is, for now, an admittedly weak way to provide a mechanism for dialogue language switching during an iteration. If the X character is not given, ITER simply reports what language ("English", "French", etc) is current. Otherwise, the X must immediately follow the 'L' (or 'l') with no intervening delimiter. (Sorry, we don't have a real command language here - it's the hammer and chizzle type (brute force method) so far). Only specific Xs are recognized and if the actual

X isn't one of these, an error message is issued and the current language is not changed.

See the FEEDBACK section for a discussion of multi-language dialogues, adding new languages, and etc.

? -To find out what responses are allowed.

For UNIX compatibility, this response is accepted as a way to explicitly ask for the internal HELP screen-full. This text is meant as an aid to someone who already understands the basic operation of ITER; it is an on-line suppliment to the ITER tutorial (this document) and is not meant as a replacement for it. The "help text" is always presented in the current dialogue language; to see this text in a different language, you must first switch languages (see 'L') and then ask for the help.

You get this same "help menu" if you type any invalid response to ITER, but it is preceded by a message to the effect that the response was invalid. The '?' command legitimizes the requisition of help.

You will notice that the "help text" display changes from time to time. This is because ITER takes into account current defaults, etc, when reporting what is legal. For example, the '//' (research) command is not legal until you have already done a search in either direction, so it is simply not shown on the "help screen" until it IS legal. Aside from not misleading people, this has the added advantage of allowing ITER to display the default (what '//' would search for) as part of the menu.

NN -The NN character string must be completely numeric, and it is interpretted in decimal. ITER repositions you to exactly that element in the Name-SET. Element ordinals are in the range 1..M (where M is the cardinality of the Name-SET), but a 0 is allowed here as another way to reposition yourself to the beginning of the set. Although ITER reports "which element you are on" as, for example, '#3', (to motivate "number 3"), you never type the '#' character back to ITER. Unadorned numbers are always taken to be references to element ordinals.

If you give an NN which is outside the set, (an NN > M), you are repositioned to the last element in the Name-SET. This is for "compatibility" with various MAIL systems, etc, in which one is accusomed (presumably) to ask for '9999' as another way to get to the end. See the more ED-inspired '$' and '^' commands, below, for a more civilized approach to this.

+NN -To reposition ahead. The '+' sign must be given, here, and the Ns must be decimal numeric. The default NN string is 1 so entering just '+' will move you forward by one element (just as 'N' currently does, but this may change). The effect is to move you ahead in the Name-SET exactly that many elements. If you give a +N which would have the effect of taking you outside the range of the last element, ITER produces an error message and your position in the set remains unchanged.

$ -To reposition to the last element of the Name-SET.

-NN -To reposition backwards. The '-' sign must be given, and the Ns must be decimal numeric. The effect is to move you backwards in the Name-SET exactly that many elements. The default NN string is '1' so entering just '-' will move you backward by one element in the Name-SET. If you give a -N which would have the effect of taking you outside the range of the first element, ITER produces an error message and your position in the set is unchanged. Effectively saying "GOTO 1" or "GOTO 0" in this way is equivalent and allowed.

-To reposition to the first element of the Name-SET.

[x] -To ask for the Name-SET to be displayed. If you enter just '', the set is displayed exactly as it exists (in ITER's memory). If you give 'O' as the optional [x] character, (for "ordinals"), the set display is right-shifted just enough to let ITER precede each non-comment line of the set with the element ordinal number. These are exactly the same ordinals that you can use to "select" specific places (to relocate your iteration to) in the set.

A [x] qualifier of 'V' (for "verbose") is also allowed, and an experimental display is produced which gives you even more information. I may withdraw this "feature".

v -For 'Verbose'; used to toggle the "verbose flag". That is, if you are in VERBOSE mode, a 'V' command will put you into TERSE mode where informational messages are suppressed. If you are already in TERSE mode (the default), a 'V' command will put you into VERBOSE mode.

In verbose mode, ITER is a lot more chatty than it is in TERSE mode. In verbose mode, ITER always displays the command line just before it is passed off to a subprocess shell. Moreover, in query mode there are numerous informational messages that remind you of what was done as the result of a command you entered. If the mode is also TERSE, these messages are suppressed while in VERBOSE they are not.

=[v] -The '=' command is patterned after ED's "tell me where I am" counterpart. If you enter just '=', the current Name-SET line is displayed with its associated ordinal.

For lack of a better place to put it, a '=V' function is also provided. This one is like a "tell me everything you know" command. A half-screen-full of information is displayed in which ITER tells you the current version number (it's 1.0 at the time of this writing), the current iteration and !-command templates, the setting of the VERBOSE vs TERSE flag, the exit status returned by the last subprocess initiated by ITER, and etc. I intend on adding to this display as the need arises.

If you find youself "lost" during an iteration, try '=V'.

/xxx/ -Search forward. This command, also modeled after ED, lets you reposition yourself forward in the Name-SET based on the content of the set. You give any ascii string, and ITER does a match on each element in the set starting with the one after the current one. Each

match is case-insensitive, that is, both the search argument and the Name-SET line are mapped to upper case before the comparisson is made. There is currently no way to do a case-sensitive match.

A match is found whether the given (or implied) 'xxx' argument exactly matches the Name-SET element, or if the substring appears anywhere within the set element. In verbose mode the "element found" message is slightly different to let you know that you may have matched to a substring when you may have been looking for a specific element.

If the last line of the Name-SET is encountered and no match found, ITER reports an error message and your place in the set remains unchanged.

If you have already done a search either forward or backwards, you can default the 'xxx' string to null and "search again" for the remembered argument. This is useful for switching direction - the same default argument is shared by // and ??.

?xxx? -To search backwards. This behaves exactly like the /xxx/ command except that ITER starts with the Name-SET element before the current one, and stops after it has tried a match with the first element in the set.

There is no way to do a "wrap-around" search. However, if you search in one direction and don't find what you're looking for, you can default the search argument but switch direction to look elsewhere in the set for the same thing. If you don't find it then, the string is either in the current element or it is not in the set at all.

If you are in query mode and want to do something other than what is provided for above, you may find it useful to "push down" ITER and enter into a completely new level of SH; somewhat like starting a new terminal session. To do this, use the <exclamation point> query mode command and explicitly give 'sh'. When you "log out" of this shell, you will return to the iterator just as if you hadn't left.

White space is removed at the beginning of all query mode commands except 'sh escape' which is always a pass-thru exception. Either upper or lower case commands are accepted and considered equivalent in all instances.

In addition to the syntax described above, you can always follow a query mode command with the UNIX-inspired '&' character to leave query mode **but remain in ITER**. For example, if you start up an iteration in non-query mode and an error occurs, you will automatically enter query mode and have a chance to fix the problem. You can then say 'y' to retry the command that caused the problem. After that command finishes, though, you will still be in query mode and ITER will present to you the next instantiated command.

To say "yes" to that and to also leave query mode, simply respond:

y &

ITER will execute the indicated command, and check the exit status (as always) when it completes. If no problem is indicated, ITER will continue on with the

next element in the Name-SET as though query mode had never been entered in the first place.

In all cases, ITER leaves query mode only if there is not an error in the associated response. For example, giving "/next/ &" will NOT leave query mode if "/next/" is not found within the remaining part of the Name-SET.

When query mode is successfully left in this way, the semantics are (literally) to go back to iterating in the usual way, with the default response ('y') being applied behind the scenes each time around the loop.

When the end of the Name-SET is reached, the action taken by ITER depends on query mode. If you start up an iteration which runs without user interaction, ITER will finish quietly and return control to the invoking shell. On the other hand, if you are in query mode for any reason when the end of the set is reached, ITER will refuse to "run off the end". That is, a message will be produced, and the final instantiated command will be presented to you as usual. To leave immediately, simply use 'x' or 'q'.

If you want to "run off the end" of an iteration and are in query mode, you must arrange to NOT be in query mode when the end is reached. Simply use '&' to leave query mode, which works conveniently with the 'n' response as well as with the rest. This issue of "hanging on" in query mode is treated in FEEDBACK, below.

Background iterations are immune to signals so you cannot effectively use query mode when you select "unattended operation". (Indeed, UNIX will let you do this, but you get the same intermixed I/O confusion that happens whenever a background process competes with a foreground process for user interaction via the terminal.)

9.5 Name-SETs

A set is a collection of elements, and an element is (currently) defined as any string not containing a newline and not beginning with the colon character. This includes the <empty> string, which is indeed the "empty element".

Thus:

```
:
: This is a documented Name-SET
:
first element

last element
```

is a sample Name-SET whose cardinality (number of elements) is 3. Note that the middle element in this set is empty - the null line is still a valid element. The "comment lines", however, are simply "not there" as far as the iteration is concerned.

ITER operates by first loading the indicated Name-SET into memory - be it by reading standard input until end of file, or by reading the indicated (named) file. In this sense, ITER is not strictly a pipeline, but this "problem" could be fixed. It is nice, at least in query mode, that when you ask to have the Name-SET displayed, you really do get to see all of it - not just all of it so far.

9.6 Nested Iteration

If the command template that you give in an iteration is yet another ITER command, you enter what I term a "nested iteration". This is a natural side effect of the fact that ITER does no special-casing whatsoever on the commands that it invokes - if the UNIX shell will execute a command, ITER can iterate it.

Nested iterations are usually based on having a Name-SET which contains the names of other sets. For example, in developing ITER it was natural for me to maintain a Name-SET of the various pieces (files) that are used to build ITER from source, test it, etc. Thus, the 'header' files (the C *.h files) were all named in a set, and I had two other sets that allowed me to separate out the C sources used in the 'support' functions versus the 'body', respectively. The set called 'system', then, contained the names of these Name-SETs.

There were numerous things that I used these sets for during the development of ITER. Once of them was SAVE, a command file that I wrote to squirrel away a /tmp copy of a file whose name it was passed as the first parameter. At each baselevel, I used SAVE to take a snapshot of my sources by doing the following nested iteration:

```
$ iter -s system it -s %e save
```

The ITER level feeds the IT level with the names of the sets, while the IT level repeatedly calls SAVE with the names of the files that I saved away. See FEEDBACK (19) where I point out some problems in differentiating the %Es that belong to each level. Just now, all the %Es that you give on the original command line are considered to belong to the outer iterator since the inner command line is instantiated quite blindly.

In fact, I also had a set called 'etc' which contained the names of all the other files that had anything to do with my ITER 'system'. To really save away the whole thing, I did:

```
$ cat system etc | iter -v it -s %e save
```

I found it very useful to have one level in a nested iteration speak french, and the other level speak english - especially in query mode. In that way, it's very hard indeed to "get lost".

9.7 SEE ALSO

sh(1), csh(1), find(1), grep(1)

Also, see VME's commands called EXEC, ITER, and ESP. VME stands for "Vax Made Easy", and is the tool set where the notion of iteration was first prototyped (on VAX/VMS).

9.7.1 Bugs or Limitations

If you interrupt a nested iteration, both ITERs receive the signal and note this fact. Only the innermost ITER appears to get control, though, and it will prompt in the usual way. If you subsequenly do an X from the innermost ITER, the outer one will regain control in the expected way, but it will issue a message to the effect that there was an error or interrupt in the iterated command. I'm not exactly sure what to do about this one. I'd like to make the outer ITER unaware of what is rightfully the busisness of the inner one.

It's not clear to me that background ITERs are indeed immune to interrupt in all cases. I need to test this more thoroughly...

9.8 Feedback on ITER Futures

In this section I will present several "open issues" regarding ITER for the purpose of solicitng feedback on these design or functionality issues.

1) I hate the fact that you don't get a message when you don't give a Name-SET and you don't redirect the input. In this case ITER just sits there like a dummy waiting for you to type in the Name-SET by hand. I have tried to write C code to detect this case, but can't seem to make it work. It's a matter of being able to distinguish when ITER is reading from standard input (that's easy), and when, in this case, the standard input is a pipe instead of a terminal (that's hard). All this in the name of UNIX compatibility.

2) The notion of "advance +1" is somewhat limited because it is always to the next element, in the forward direction, that you go after a successful "yes".

   My proposal is that there is always an "advance clause", and that ITER use this when "proceding" each time after the ITER fundamental operation succeds. In default, this clause is "advance +1", but you should be able to set this in query mode (and otherwise?) to be, for example, '/next/' or '?name?' or '-1', etc. The semantics of this would be just as if that advance clause were issued by the user instead of the now-default '+1' string which tells the iterator how to reposition for the next instantiation. Setting the advance clause to '0' would let you "sit in an iteration" just looping. In my experience, there are numerous reasons for such a feature; it's essentially "repeat" or "while(true)" or however you want to express that.

3) The default now is for ITER to read the Name-SET from standard input. Perhaps it's just that I'm used to having this otherwise (on VAX/VMS), but nevertheless I find myself giving:

iter -v name cmd1 cmd2 ...

the whole time, and ITER just sits there. Why? Because it's waiting for me to type in the set directly. I hardly **ever** want to type in sets directly, so therefore find this extremely annoying - especially in light of the fact that all you can do when you realize what's gone wrong is to quit and retype the entire command.

Should I change this default so that the first word on the command line is assumed to be the name of the set, and you have to say something else to **not** have this happen?. If so, the -I (or whatever) option flag would be required when you **do** want ITER to read the set from standard input. This would mean that:

ls *.c | iter -q cmd1 cmd2

would read the file 'cmd1' as the set, and, assuming that it didn't make a fool out of itself doing that, it would then ignore the piped-in data and ask you if you wanted to iterate a command that would look strange to say the least. If you didn't give the -Q option on that, the iterator could conceively go on to do disastrous things - but then, UNIX is like that - it let's you "do anything".

Seems like you can't have your cake and eat it too.

4) Is there any point in allowing an iteration over the empty set? It's easy to detect; but I wonder if it's worth the effort. This is related to (7). If you can change the set once you're speaking to ITER in query mode, then you don't have to retype the command template if all you forgot to do was to specify the set correctly.

5) And what about "save" And "restore" ? It might be nice to be able to completely save the context of an iteration (in a file?) for the purpose of coming back to it again. One problem with this is that of deciding just how much context to save? I tend to think it should be all or nothing - this includes the defaults for everything that is created dynamically, as well as where you are in the Name-SET, and etc.

6) We could provide for ITER to let you have your own shell or editor by using shell variables to pick up the pathname of the process that gets invoked, much like CSH does. Is this worth it? The scenario would be that the user would have to define something like SHELL (or we use the existing convention if one can be found) to have his "own shell" invoked to do ITER's fundamental operation. I presume that this same shell would be used to implement exclamation mark <cmd>s, too.

Similariliy, the shell variable EDITOR could be used to pick up what we now default to '/bin/ed', and a real editor could be used to edit the command template. If we ever allow the Name-SET to be edited, this will be a more pressing concern.

7) Should we allow the user in query mode to change the Name-SET? If so, does this actually change the file that the set may have come from? Is there a connection with this and the propose "save and restore" features (above)?

8) There is no way to make a /xxx/ or a ?xxx? search look at the current line. You have to move back one and do a "fake" search to see if there is a match. Does this matter?

And what about wrap-around searches and case-sensitive searches?

9) I find it very annoying when I'm in query mode, do an 'n' because I don't want to work on this element, and then ITER refuses to exit when it finds out that I have exhaused the Name-SET. Even a subsequent 'n' will not get you out - you have to leave query mode or use 'x' or 'q'.

How can we fix this "anomaly" while retaining the saftey margin provided by ITER's tendency to want to "hang on".?

10) Nested iterations know absolutely nothing about each other. This is why the '=V' function can't tell you what level you are at. It should be able to, I think. It's certainly something that one needs to know, and having to "work it out" by looking at the various stuff that '=v' tells you is primitive to say the least.

11) There is always the question of having a "real command language" for ITER. Admittedly, this 'y' and 'x' stuff is very UNIX-like, but even I find it annoying when I say "yes" and get told that this is "unknown or invalid". Is it worth using YACC and putting in a real keyword-driven language?

Once we have a mouse and decent CRT interface, this issue will go away. A mouse will help greatly in being able to pick elements out of the set, too. See FEEDBACK (24) for more details.

12) Should the names 'e' and 'x' be synonymous? Just now there is no syntax support for "exit". While we're at it, are 'x' and 'q' named properly or do we have it backwards? Is there anything like a UNIX standard for this?

13) The toggling of "the VERBOSE flag" seems somewhat hacky, although it's exactly the model that we're seeing more and more in menu-driven systems. Is it worth inventing a command to "set terse" and "set verbose"?. Should there be intermediate levels of "verbosity", while we're at it?. And what about the other dimension of user tailorability: the NOVICE versus EXPERT distinction that is often (sometimes?) made in highly interactive tools? I wonder if some of the query mode behavior has its roots in novice versus expert user assumptions...

14) The 'c' command lets you edit the ITER command template, but forces you to go thru a rather elaborate procedure to do it. Should we provide for one to simply re-enter the template - ie. something like 'C*' (the '*' meaining "all") in which ITER simply prompts for the user to retype the whole command template. This would completely avoid the ED phase, which is convenient especially if you do want to respecify the template. If we do this, should we do the automatic %E defaulting that ITER does on the command line template, or should we force the user to remember to put in the %Es, etc?

15) The whole idea of having a "multi-language dialogue" is suspect. Is it worth it at all? I believe that my prototyping shows that it's practical to have several languages, and that from a programming and methodology point of view, the "feature" adds only marginal complications. However, there are several unanswered questions...

What should we do about the "names" for these dialogue languages? The names for a language change when your base dialogue does; does this mean that the allowed operands change when you switch languages?.

This same problem can be seen in the 'y' versus 'o' (for "oui") query mode response where I am trying to "have it both ways". How far should we persue this notion?

I'd like to experiment with adding more dialogue languages; anyone speak German or Spanish well enough to supply the message file translations?

16) There is no default query mode command, mostly because I'm leary of going over board with too much defaulting. I am tempted to define '<cr>' as an equivalent for "yes" or "advance", but it's a pain to have the iterator jump ahead on you if you hit <cr> inadventently.

How about a user-defined way to set this default? In this scenario, '<cr>' would not ordinarily have a meaning, and if you hit <cr> you would get a reminder of the fact that you can define it to have a meaning. If you do make this definition, then you can "set up" '<cr>' to imply whatever semantics you like. This might be a pain to have to do the whole time; does this imply something like inheriting this definition from the environment, like SHELL and EDITOR, etc?

17) There is no default '&' command either. How does this relate to (16)?

18) I talk about "help text", etc., but there is no such thing (now) as a help command. If you ask for 'h', you get a message about this being an invalid command. Admittedly, you do get the help screen-full in any case, but I wonder if we should really have a command other than '?'. Is there a UNIX standard in this area?

19) When you really do a nested iteration, you almost always end up using the form where the %E is defaulted for the innermost iteration. See the SAVE example, above. If I had wanted to use something like my "CP" example for the inner command template, in the SAVE example, the outermost ITER would obscond the %Es and I'd find that I just couldn't do what I wanted to. How do we rectify this situation, if at all?

20) There is very definitely an inconsistency between +NN and NN with respect to the boundary conditions. I am not sure whether one is doing the user a favor to automatically reposition you to the beginning or end when you say something suspsect like 9999, or, even more suspect, '-10' when you are on element number 3. On the other hand, it doesn't seem very "user friendly" to be extremely ridgid about it, either. Should the semantics of the various NN-commands change with respect to Ns out of bounds?

21) Now that ITER has been adopted by Sol, should it be rewritten in PASCAL?.

22) It seems that having ITER stop and enter query mode every time there is an error is NOT always what you want. Sometimes, especially when capturing the output of an iteration, you really do want it to charge on whether or not there is an error. A particularily blatant example is GREP. If you use GREP to look for a string in a set of files, an error return is taken by GREP for each file where the string was not found. This causes the iterator to take control as if something unexpected had happened, which is almost never what you wanted in this case.

How can we provide this "continue-on-no-matter-what" feature? I suspect that it may be related to the "advance" notion - this one is like a "force advance", or something. See the discussion of "advance" in FEEDBACK (2) above.

23) In the VAX/VMS prototype, there is very definitely a tightly-coupled interface between "iter" and other command-level utilities. That is, you can write a user interface which has the iterator built in as an inherent part - or, more appropriately put- you can provide something which gives this appearance. In fact, the iterator is not built in at all; it's just that one can change its exterior characteristics and interface to it in such a way that, for example, the FLIP electronic book library iterator is seen as a tool which uses iteration as an inherent part, even for users who know nothing about iteration, ITER, or any of the underlying facilities. How far should we go in this direction with the UNIX prototype?

I think this is equivalent to the question: should we make ITER "callable" in the sense that people are making editors "callable" these days?

24) With the advent of multi-window systems, bitmap screens, and the like, one wonders how ITER fits into all of this.

My contention is that we should indeed have a bitmap-specific version of ITER. One part of the screen could be used to continually display, scroll, and select from the Name-SET; while the other "window" would be where commands actually get exectued. Comments?

9.9 Further Information

For on-line information on using ITER, see the "menu" display that the '?' command produces. Also, for hard copy info, you can get a printed version of this text by copying or printing the ASCII file:

/numelec/iter/src/iter.doc

or by contacting the author.

Any suggestions on how to improve ITER or this text would be appreciated. Contact: Kevin Pammett, INRIA, Rocquencourt. (3) 954-9020 poste 160 or 523.

[end of ITER tutorial text -- updated as of 25 November, 1983]

This page left blank for double-sided copying

Table of Contents

Imprimé le Jeudi 29 Mars 1984

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique